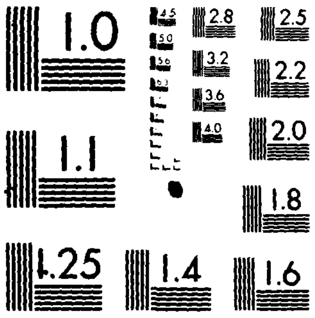
AD-A116 161   MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
MODEL PROGRAM GENERATOR: SYSTEM AND PROGRAMMING DOCUMENTATION, --ETC(U)
MAY 82 K LU   N00014-76-C-0416

UNCLASSIFIED   NL

1 OF 2


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(2)

Technical Report

MODEL PROGRAM GENERATOR: SYSTEM AND PROGRAMMING DOCUMENTATION

Spring 1982 version

by

Kang-Sen Lu

UNIVERSITY of PENNSYLVANIA
The Moore School of Electrical Engineering
PHILADELPHIA, PENNSYLVANIA 19104

82 06 11 ~~103~~
40

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD A116161	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Model Program Generator: System and Programming Documentation		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER Moore School Report
7. AUTHOR(s) Kang-Sen Lu		8. CONTRACT OR GRANT NUMBER(s) N00014-76-~~0~~C-0416
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Pennsylvania, Moore School of Electrical Engineering, Department of Computer Science, Philadelphia, Pennsylvania 19104		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-153
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program, Code 437 Arlington, Virginia 22217		12. REPORT DATE May, 1982
		13. NUMBER OF PAGES 159 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

General Distribution

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Automatic Program Generation, Non-Procedural Languages, Model, Very High-Level Languages, Program Efficiency

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document describes the algorithms and mechanisms of the MODEL Processor which is a software system performing a program writing function. It also documents the program structure and procedures of the Processor. The MODEL Processor has been designed to automate the program design, coding and debugging of software development, based on a non-procedural specifications of a program module in the MODEL language. A program module is formally described and specified in the MODEL language, whose statements are then submitted to the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Processor. The set of MODEL statements describing a program module is referred to as a specification. The Processor, performs the analysis (including checking for the completeness and consistency of the entire specification), program module design (including generating a flowchart-like sequence of events for the module), and code generation functions, thus replacing the tasks of an application programmer/coder. The Processor's capability to process a non-procedural specification language is built on application of graph theory to the analysis of such specification and to the program generation task.

Another important function of the Processor is to interact with the specifier to indicate necessary supplements or changes to the submitted statements.

The Processor produces a complete PL/1 program ready for compilation as well as various reports concerning the specification and the generated program. The Processor output reports include a listing of the specification, a cross-reference report, subscript range report, a flowchart-like report of the generated program, and a listing of the generated program.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

OVERVIEW

This document describes the algorithms and mechanisms of the MODEL Processor, which is a software system performing a program writing function. The MODEL Processor (hereafter called the Processor) has been designed to automate the program design, coding and debugging of software development, based on a non-procedural specifications of a program module in the MODEL language. As shown in Figure 1.1, a program module is formally described and specified in the MODEL language, whose statements are then submitted to the Processor. The set of MODEL statements describing a program module is referred to as a <u>specification</u>. The Processor, performs the analysis (including checking for the completeness and consistency of the entire specification), program module design (including generating a flowchart-like sequence of events for the module), and code generation functions, thus replacing the tasks of an application programmer/coder. The Processor's capability to process a non-procedural specification language is built on application of graph theory to the analysis of such specification and to the program generation task.

Another important function of the Processor is to <u>interact</u> with the specifier to indicate necessary supplements or changes to the submitted statements.

The Processor produces a complete PL/1 program ready for compilation as well as various reports concerning the specification and the generated program. The Processor output reports include a listing of the specification, a cross-reference report, subscript range report, a flowchart-like report of the generated program, and a listing of the generated program, all to be described fully later.

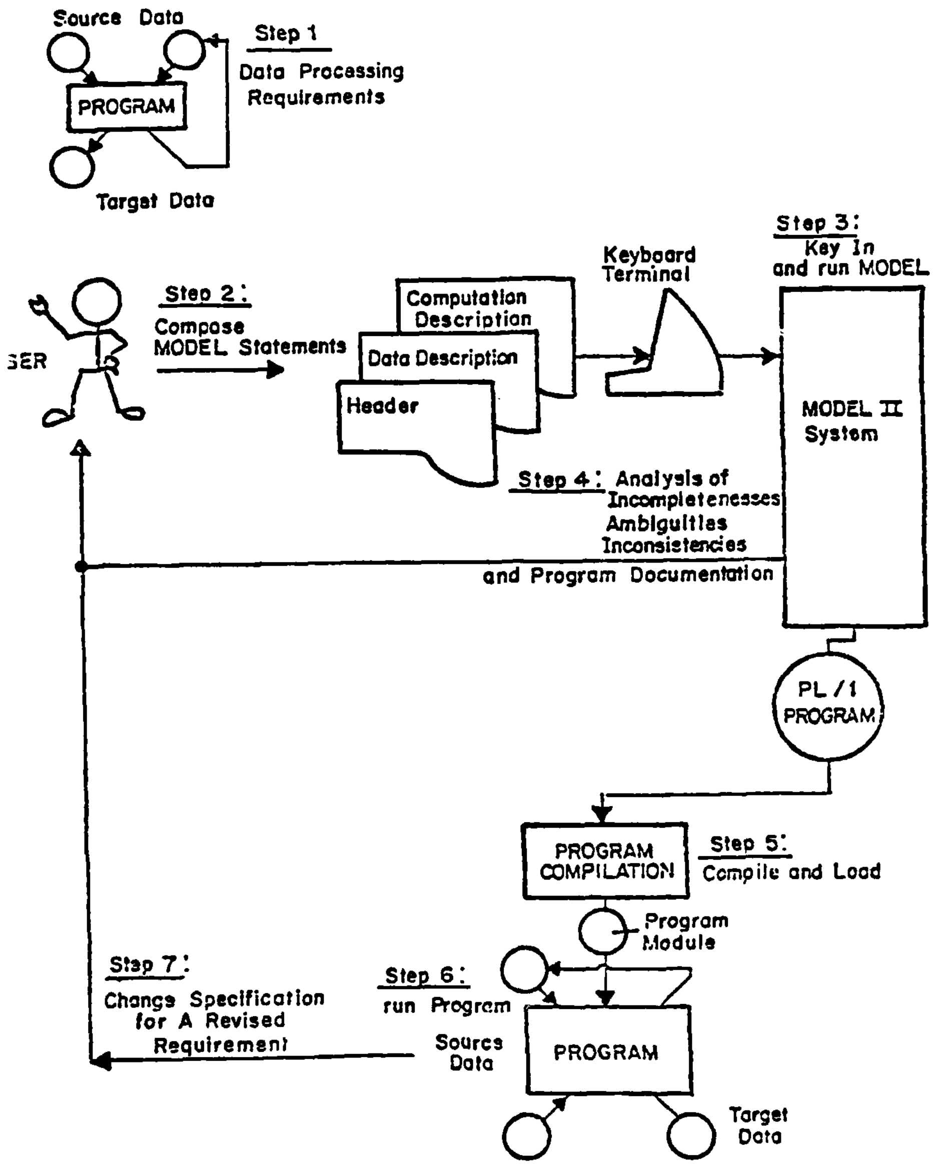


Figure 1.1 The Overall Procedure For Use of MODEL

Processing of a specification written in MODEL by the Processor consists of four phases shown in the system flowchart of Figure 1.2, which is the first refinement of Figure 1.1. Some of these phases represent adaptations of known but state-of-the-art technology, while other phases involve more novel innovations in analysis of the specification and in the design and code generation for the application program.

Each of the four phases depicted in Figure 1.2 is discussed below.

Phases 1: Syntax Analysis of the MODEL Module Specification

In this phase, the provided MODEL specification is analyzed to find syntactic and some semantic errors. This phase of the Processor is itself generated automatically by a meta-processor called a Syntax Analysis Program Generator (SAPG), whose input is syntax rules provided through a formal description of the MODEL language in the EBNF language (yet to be discussed). In this manner, changes to the syntax of MODEL during development can be mode more easily.

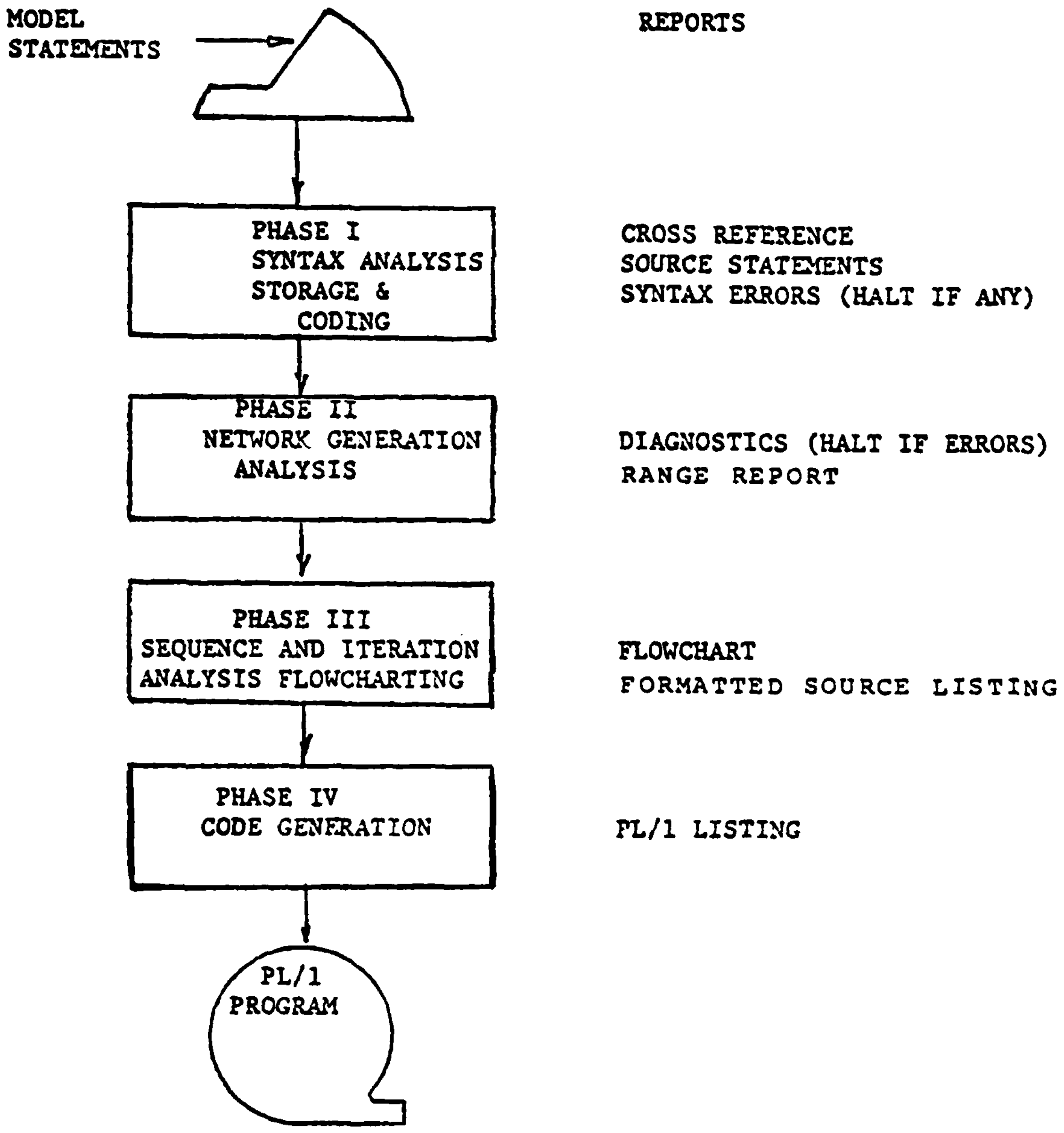

MODEL STATEMENTS
REPORTS
PHASE I SYNTAX ANALYSIS STORAGE & CODING
CROSS REFERENCE
SOURCE STATEMENTS
SYNTAX ERRORS (HALT IF ANY)
PHASE II NETWORK GENERATION ANALYSIS
DIAGNOSTICS (HALT IF ERRORS)
RANGE REPORT
PHASE III SEQUENCE AND ITERATION ANALYSIS FLOWCHARTING
FLOWCHART
FORMATTED SOURCE LISTING
PHASE IV CODE GENERATION
PL/1 LISTING
PL/1 PROGRAM

Figure 1.2 Phases of the MODEL II Processor

A further task of this phase is to store the statements in a simulated associative memory for ease in later search, analysis, and processing. Some needed corrections and warnings of possible errors are also produced in a report for the user. Also, a cross-reference report is produced.

A description of the syntax and statement analysis phase is covered in detail in Chapter 3.

Phase 2: Analysis of MODEL Specification

In this phase, precedence relationships between statements are determined from analysis of the MODEL data and assertion statements. Ths specification is analyzed to determine the consistency and completeness of teh statements. Each MODEL statement may be considered to be an independent stand-alone statement. The order of the user's statement is of no consequence. However, in analysis of the statements, precedenece relationships are determined based on statement components. These relationships are used to form the nodes and directed edges of an array graph (yet to be discussed) on which completeness, consistency, ambiguity, and feasibility of constructing a program can be checked. Various omissions or errors are corrected automatically, especially in connection with use of subsctipts. Reports are produced for the user indicating the data, assertions, or decisions that have been made by the Processor, or contradictions that have been found. In addition, a report showing the range of each subscript is generated.

Explanation of this process is covered in Chapter 4 and 5.

Phase 3: Automatic Program Design and Generation of Sequence and Control Logic

This phase of the Processor determines the sequence of execution of all events and iterations implied by the specification, using graph theory techniques. It determines also the sequence and control logic of the desired program. The result of this phase is a flow of events, sequenced in the order of execution. Thus, the output of this phase is similar to a program flowchart-like report. At the end of this phase it is also possible to produce a formatted report of the specification.

This phase is presented in detail in Chapter 6.

Phase 4: Code Generation

AT this point in the process it is necessary to generate, tailor, and insert the code into the entries of the flowchart to produce the program. In particular, read and write input/output commands are generated whenever the flowchart indicates the need for moving records. The assertions are developed into PL/1 assignment statements. Eherever program iterations and other control structures are necessary, program code for them is generated. Declarations for object program data structures and variables are generated. Code is also generated for recovery from program failures when bad data is encountered during program execution. The product of this phase is a complete program in a high level language, PL/1, ready for compilation and execution. A listing of the generated program is produced.

The remainder of this report expands on the above mentioned phases. Chapter 2 discusses the syntax and semantics of each type of MODEL statements. Figure 1.3 provides a tree diagram of the major modules. The name of the modules in this diagram are referenced throughout the remainder of this report wherever the corresponding task is explained. As seen at the top of Figure 1.3, a MONITOR governs the execution of the different phases of the Processor, and does not allow succeeding phases to proceed without the success of the previous phases. At the second level of Figure 1.3, the major phases of the Processor are named (1) SAP (Syntax Analysis Program), Chapter 3; (2) NETGEN (Network Generation) and NETANAL (Network Analysis), Chapter 4 and 5; (3) SCHEDULE (Schedule events and generate flowchart), Chapter 6; and (4) CODEGEN (Code Generation), Chapter 7. Below this level of Figure 1.3, the diagram shows the names of the modules subordinate to each of these phases. Each of these subroutines is discussed throughout this report.

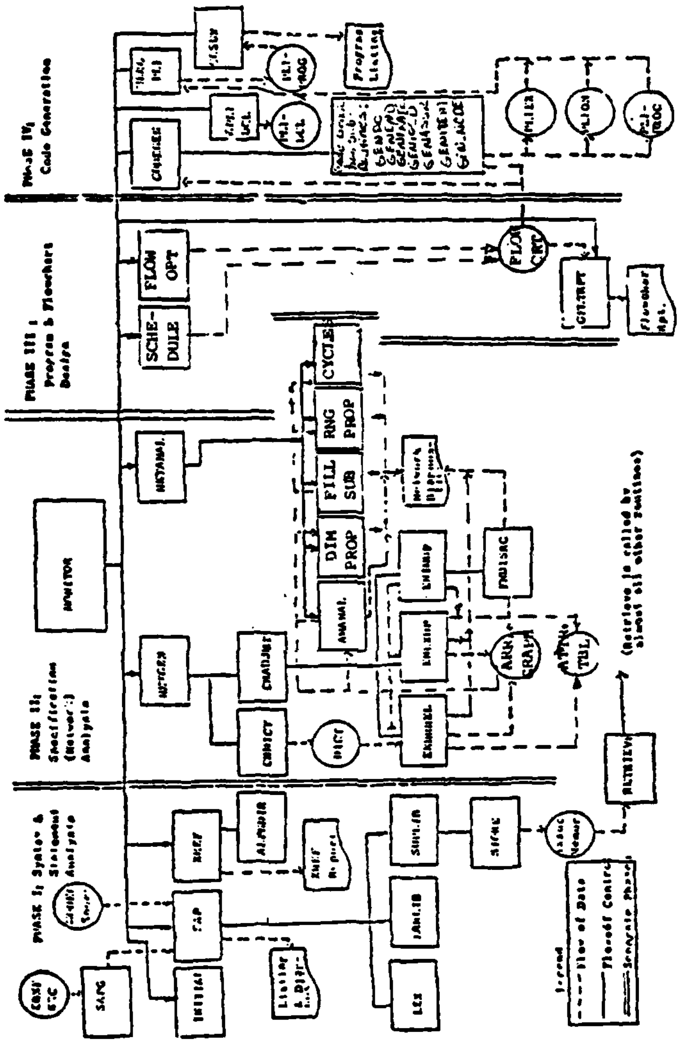


Figure 1.3 Major Modules of the MODEL Processor

CHAPTER 2

SYNTAX AND SEMANTICS OF THE MODEL LANGUAGE

2.1 STRUCTURE OF A PROGRAM SPECIFICATION

A program specification written in the MODEL language consists of three major parts: program header, data description, and assertions. The program header specifies the name of the program and the external files which store the input or output data of the program. The data description statements are used to specify the data structure of the input or output files and the structure of the intermediate results. The assertions are used to define the values of the intermediate or output variables specified in the data description statements. Although the user is encouraged to group statements together and order the parts in the sequence mentioned above, the statements in a program specification can be put in any order, i.e. the order of the statements is irrelevant to the meaning of the specification. That is one reason why we call MODEL a non-procedural programming language. In this section we discuss the statements in the program header. We will discuss in section 2.2 the data description statements, and in section 2.3 the syntax and the semantics of the assertions. We will discuss in section 2.4 the use of control variables.

Only the basic MODEL language is described here. Short-hand and high level dialects are not described as they are always translated automatically into the basic language. The syntax rules of the MODEL statements will be defined with extended BNF notation. Identifiers enclosed by the angle brackets ('<' and '>') are non-terminal symbols. The metasymbols used include:

1. ::=, it is read as 'is-defined-by'.
2. [...], a pair of square brackets is used to enclose a string which is optional.
3. |, a vertical bar is used to separate alternatives.
4. {...}*, a pair of braces followed by an asterisk is used to enclose a string which can repeat any times (including zero).

The program header consists of three types of statements, namely the <u>module</u> statement, the <u>source file</u> statement, and the <u>target file</u> statement.

Module Statement

The syntax rule for the module statement is as follows.

```
<module-statement>::=
     MODULE : <identifier> ;
```

The user-chosen identifier is used as the name of the program being specified.

Source File Statement

The syntax rule for the source file statement is as follows.

```
<source-file-statement>::=
     SOURCE [ FILES | FILE ] : <identifier> { , <identifier> }* ;
```

The source file statement consists of a list names of files which serve as the input files of the program. The source files are assumed stored in external storage devices.

Target File Statement

The syntax rule for the target file statement is as follows.

```
<target-file-statement>::=
     TARGET [ FILES | FILE ] : <identifier> { , <identifier> }* ;
```

The target file statement lists the names of files which serve as the output files of the program. The output files are assumed to be on external storage and they serve to retain the computation result for future use.

2.2 DATA DESCRIPTION STATEMENTS

In a non-procedural programming language every variable can only have a single value. Therefore, different variable names should be declared for different data involved in the computation. The data structures in external files, or the schemata of files, can be described in MODEL with data description satatements. Logically related variables may also be grouped together as in PL/I. The user must also declare the data types of the components of a variable in data description statements. The MODEL language has been designed to relieve the user of concern for I/O control. In general, I/O can be a complicated part of a programming language. A few simple mechanisms have been included in the data description statements to ease the I/O programming task. Examples include the ability to describe file organization and to indicate a key field for direct accessing a record. In section 2.2.1 we will discuss the way to specify the data type of a variable; in section 2.2.2, the way to describe data aggregates; and in section 2.2.3, the mechanisms used for I/O related programming.

2.2.1 DATA TYPES

The smallest unit of data in a program is a field. A field may contain a datum of some type supported by the MODEL language. The available data types includes picture, character, bit string, and numbers. It is the user's responsibility to select a data type for each field.

Field Declaration Statement

The syntax rule for a field declaration statement is as follows.

```
<field-declaration-statement> ::=
     <identifier> [ IS ] <field> <data-type> ;
<field> ::=  FLD | FIELD
<data-type> ::=  <type> <leng-spec>
<leng-spec> ::=  ( <min-length> [ : <max-length> ] )
<min-length> ::= <integer>
<type>::= <pic-desc> | <string-spec> | <num-spec>
<pic-desc> ::=  <pic-type> ' <string> '
<pic-type> ::= PIC | PICTURE
<string-spec> ::= CHAR | CHARACTER | BIT | NUM | NUMERIC
<num-spec> ::= <num-type> [ <fixflt> ]
<num-type> ::= BIN | BINARY | DEC | DECIMAL
<fixflt> ::= FIX | FIXED | FL | FLOAT | FLT
<max-length> ::= <integer>
```

A character string may be of fixed length or variable length. For a fixed length character string the length in byte units should be specified in the type declaration. A variable length character string is specified through declaring the range of the possible length of the string. When a field X of variable length string occurs in an input file, its length should be specified by an associated control variable called LEN.X.

Example:

```
     A IS FIELD CHAR(6) ;
     B IS FIELD CHAR(0:10);
```

The field A is a string of six characters and the field B is a variable length character string with maximum length ten. The actual length of the field B should be specified by a control variable called LEN.B in some assertion.

The available operations for manipulating character strings include lexicographic comparison, concatenation, and extracting substring. The discussion for the character string is also applicable to the bit string data type.

The data types for numeric data include picture, floating point decimal, floating point binary, fixed point decimal, and fixed point binary. The operations applicable to numeric data are arithmetic operations, comparison, and conditional definition. It should be noted that the picture and character typed variables have a printable representation. Therefore, it is suitable for data contained in reports. Other numeric data types are generally used for the data stored in the computer system. The PL/I target language incorporate extensive type conversion and therefore the user is generally relieved of this concern.

2.2.2 DATA STRUCTURES

Usually there are two ways to group logically related data together to form data structure. An array contains homogeneous data elements and a structure contains heterogeneous data elements. In MODEL a generalized data aggregate can be used to specify arrays and structures. The data aggregate is called a group or a record in MODEL language.

Group Declaration Statement

The syntax rule for the group declaration statement is as follows.

```
<group-declaration-statement> ::=
     <identifier> [ IS ] <group> ( <member-list> ) ;
<group> ::=  GRP | GROUP
<member-list> ::= <member> { , <member> }*
<member> ::= <identifier> [ ( <occspec> ) ]
<occspec> ::=  * | <minocc> [ : <maxocc> ]
<minocc> ::= <integer>
<maxocc> ::= <integer>
```

In the group declaration statement an identifier is declared as a data group which contains a list of members. Each member may optionally repeat some number of times. If a member repeats, it is considered as an array of one dimension more than the group containing it. There are three ways to specify the number of repetitions over a dimension of an array. If the number of repetitions is a constant, then the constant can be specified along with the array name. When the number of repetitions is not fixed but the user knows the maximum of it, he can specify a range for the number of repetitions in the group statement. If the user does not know the maximum, i.e. where the maximum is an unknown large value, he can denote the range by an asterisk. When the number of repetitions is not a constant, it can be defined through some control variables with keyword prefix such as SIZE or END (refer to section 2.4) or definition may be omitted if it can be detected based on an end-of-file indication.

The members of a data group can be fields, or some other data groups. A data group may be declared as an array of arrays. In order to reference a unit datum of it, the user has to supply as many subscripts as the number of array dimensions. Thus the member field becomes a multi-dimensional array.

Example:

```
   A IS GROUP (B, C(10)) ;
   B IS FIELD CHAR(6) ;
   C IS GROUP (D(5), E(1:50), F(*)) ;
```

where identifier A is declared as a data group containing two members B and C. Let us assume that A is a zero dimensional variable. Since C repeats, it is a one dimensional array. Identifier C contains three members, D, E, and F. The member D repeats five times, and the member E may repeat a number of times from one to fifty. The member F has a unknown number of repetitions, so an asterisk is specified as its number of repetitions. All the members of data group C are two dimensional arrays.

2.2.3 I/O RELATED DATA AGGREGATES

In a MODEL specification, the user describes the structures of the data files with data description statements. The MODEL processor generates I/O statements automatically for the source and target files of the program based on the information in data description statements.

The record declaration statement is syntactically similar to the group declaration statement. The only difference is that the keyword GROUP is changed to RECORD. A record corresponds to a unit of data which can be physically transferred between external file and main memory.

The file is the highest-level data structure which could be declared in a MODEL specification. It is not allowed to have a structure above the file. A file structure may consist of substructures declared with group, record, or field statements. A well structured file declaration will have the file entity on the top level. Its immediate descendants (i.e. members) can be declared either as groups or records. The groups may contains groups, records, or fields. Finally on the lowest level in the file structure the data should be declared as fields.

<u>File Declaration Statement</u>

The syntax rule for the file declaration statement is as follows.

```
<file-declaration-statement> ::=
     <identifer> [ IS ] FILE [ NAME ] <file-desc>
       ( <member-list> ) ;
<file-desc> ::=
     [ KEY [ NAME ] [ IS ] <identifer> ]
     [ ORG [ IS ] <org-type> ]
<org-type> ::= SAM | ISAM
```

A file may have the KEY attribute specified. In that case, the records in the file are accessed by a part of the record contents. If a file is keyed, there can only be one record type in the file structure and one of the field in the record should be declared as the key for accessing the record. Two types of file organization are supported by the MODEL language, namely the sequential files and the index sequential files. A record in an index sequential file can be accessed faster than in a sequential file if direct accessing is necessary.

Example:

```
MODULE: MINSALE;
SOURCE: TRAN, INVEN;
TARGET: SLIP, INVEN;

TRAN IS FILE (SALEREC(*));
    SALEREC IS RECORD (CUST$,STOCK$,QUANTITY);
        CUST$ IS FIELD(CHAR(5));
        STOCK$ IS FIELD(CHAR(8));
        QUANTITY IS FIELD(CHAR(3));
INVEN IS FILE (INVREC)
        KEY STOCK$
        ORG ISAM;
    INVREC IS RECORD(STOCK$,SALPRICE,QOH);
        STOCK$ IS FIELD(CHAR(8));
        SALPRICE IS FIELD(NUMERIC(5));
        QOH IS FIELD(NUMERIC(5));

SLIP IS FILE (SLIPREC(*));
    SLIPREC IS RECORD (CUST$,STOCK$,QUANT,PRICE,CHARGE);
        CUST$ IS FLD (CHAR(12));
        STOCK$ IS FIELD(CHAR(16));
        QUANT IS FIELD (PIC'(11)Z9');
        PRICE IS FIELD (PIC'(11)Z9');
        CHARGE IS FIELD (PIC'(11)Z9');
```

2.3 ASSERTIONS

Data description statements define the data structures of the variables involved in a computation. However, the values of the variables are defined either automatically by input files or manually by assertions. Basically an assertion is an equation. On the left hand side of the equal sign there should be either a simple variable or a subscripted array name which references an array element. On the right hand side there can be any arithmetic or logical expression whose value is used to define the variable on the left hand side. The current restriction is that the assertion can only be used to define the value of a field. Operations on the higher level data structures are proposed to be translated into basic operations [PNPR 80].

2.3.1 SIMPLE AND CONDITIONAL ASSERTIONS

There are two kinds of assertions which can be used to define the value of a variable, namely <u>simple assertion</u> and <u>conditional assertion</u>. The assertions have the same syntax as an assignment statement and a conditional statement in the PL/I language, respectively. All the arithmetic and logical operations can be used in composition of expressions. In addition, the conditional expression of ALGOL language can be used in composing the expression.

Simple Assertion

The syntax rule for the assertion is as follows.

```
<assertion> ::= <simple-assertion> | <conditional-assertion>
<simple-assertion> ::= <variable> = <expression> ;
<variable> ::= <simple-variable> | <subscripted-variable>
```

The variable name on the left hand side of an assertion is called the target variable of the assertion as its value is defined by the assertion. All the variables on the right hand side are called the source variables of the assertion since their values are used to calculate the value of the target variable. In the examples shown below, a conditional expression is used to define the value of variable M.

Example:

```
  1) A = B + 5 ;
  2) X(I,J) = 4 * I + J ;
  3) M = IF OK THEN 5 ELSE 0 ;
```

Conditional Assertion

The syntax of the conditional assertion is similar to that of an IF statement in PL/I.

```
<conditional-assertion> ::=
     IF <boolean-expression> THEN <assertion>
                                  [ ELSE <assertion> ]
```

The conditional assertion has two branches, one after the keyword THEN and the other after the keyword ELSE. These two branches are selectively executed according to the truth value of a boolean expression. Since the purpose of an assertion is to define the value of a variable, there can only be one target variable in an assertion. In any case the two branches should define the same target variable. Therefore, the target variable in any branch of a conditional assertion should always be the same. It should be noted that the ELSE branch of a conditional assertion is optional. If it is omitted, the target variable may be undefined in some cases.

Example:

```
  1) IF I < 5 THEN A(I) = B(I) ;
               ELSE A(I) = B(I) + 2 ;
  2) IF END.X(J) THEN B = X(J) ;
```

2.3.2 SUBSCRIPT EXPRESSIONS

The variables used in assertions are either simple variables or subscripted variables. A specific element of an N dimensional array can be referenced with the array name followed by N subscript expressions. In the following we will discuss how the subscript expressions are formed and how they are used in composing the assertions.

Subscript expressions are composed of ordinary variables, subscript variables, and constants with arithmetic operations. The subscript variable is a special kind of variable. It does not have structure and it does not hold one specific value. Instead, a subscript variable assumes integer values in a range from one up to some positive integer. If the range for a subscript variable is fixed in the whole program specification, then the subscript variable is called a global subscript. On the other hand, if the range for a subscript variable is to be determined for each assertion, the subscript variable is called a local subscript. There are ten system predefined local subscripts named SUB1, SUB2, ..., up to SUB10. There are two types of global subscripts. One of them has the form of qualifying the name of a repeating data structure prefixed with the keyword FOR_EACH. The other is created by declaring an identifier as a global subscript with the subscript statement.

Subscript Declaration Statement

The syntax rule for the subscript declaration statement is as follows.

```
<subscript-declaration-statement> ::=
     <identifier> IS <subscript> [ ( <occspec> ) ] ;
<subscript> ::= SUBSCRIPT | SUB
```

The subscript expressions are classified into the following types according to their forms. In the following, let I denote a subscript variable, c and k denote non-negative integers, and X denote an indirect indexing vector(refer to section 4.2.2.2.) Subscript expressions may be classified as follows:

1) I,
2) I-1,
3) I-k, where k>1,
4) none of the other types,
5) X(I)
6) X(I-c)-k, where c+k=1,
7) X(I-c)-k, where c+k>1.

The range of a global subscript variable in an assertion may be declared in a subscript declaration statement. If not declared, the range is derived from an array dimension in which the subscript variable has been used in a type 1, 2, or 3 subscript expression.

Example:

```
1) I IS SUBSCRIPT (10) ;
   B(I) = A(I) ;
```

A global subscript I is declared in the subscript declaration statement and the range of the value of I is from one to ten. In the assertion, the global subscript I will assume the integer values in the range declared in the subscript declaration statement.

```
2) FACT(SUB1) = IF SUB1=1 THEN 1
                         ELSE SUB1 * FACT(SUB1-1) ;
```

The range of the local subscript SUB1 will be the same as that of the first dimension of array FACT because the subscript SUB1 occurred in the term FACT(SUB1) is in a form of type 1 subscript expression.

The use of subscript variables allows us to define all the elements of an array in one assertion. In the second example above, the whole vector FACT is defined by the same assertion.

For multi-dimensional arrays, subscripting array variables may become tedious. We have adopted the following convention to allow users to omit subscripts in array references. When all the array references in an assertion have the same leftmost subscript expression, which is a type 1 subscript and when the subscript is not otherwise referred to in the assertion, then the subscript can be omitted from the assertion systematically. For example, the following three assertions are equivalent.

```
   a1: A(I,J,K) = 2 * B(I,J,K) + C(I,J) ;
   a2: A(J,K) = 2 * B(J,K) + C(J) ;
   a3: A(K) = 2 * B(K) + C ;
```

2.4 CONTROL VARIABLES

Sometimes it is necessary to refer to attributes of the data, such as the number of repetitions, the length, or the key for accessing a record in an index sequential file. In order to allow reference to such attributes, a number of control variables are included in the MODEL language. Since the control variables are always related to some variable, they have a form of a qualified variable, with the name of the variable as the suffix and one of several reserved keywords as the prefix. In the following we will assume that X is a variable name declared in some data description statement. The control variables which can be formed from X are discussed below.

SIZE.X

If X is a repeating member of some data structure, the user can specify the range by defining the value of a control variable called SIZE.X. It should be noted that X may be a multi-dimensional array. SIZE.X defines only the range of its rightmost dimension. The ranges of the other dimensions have to be defined separately.

SIZE.X is a variable of integer type. Its value is used to specify the number of repetitions of the rightmost dimension of array X. If X(I1,I2,...,In) is an n dimensional array where I1 occurs on the most significant dimension and In on the least significant dimension, then the control variable SIZE.X(I1,I2,...,Ik) should be a k dimensional array with 0<=k<n. The first dimension of SIZE.X has the same range as the first dimension of array X, the second dimension has the same range as the second dimension of array X, and so on. The value of SIZE.X cannot be a function of any subscript Ii with k<i<=n. For every n-1 tuple (I1,I2,...,In-1) which corresponds to a possible combination of the leftmost n-1 subscripts for array X, the number of elements of array X with this tuple as their leftmost n-1 subscripts is specified by the array element SIZE.X(I1,I2,...,Ik).

Example:

```
    A IS GROUP (B(3)) ;
    B IS GROUP (C(*)) ;
    C IS FIELD ;
    SIZE.C(1) = 4 ;
    SIZE.C(2) = 2 ;
    SIZE.C(3) = 3 ;

    SIZE.C        C
    ------        ---------------------------------------
    | 4 |         | C(1,1) | C(1,2) | C(1,3) | C(1,4) |
    ------        ---------------------------------------
    | 2 |         | C(2,1) | C(2,2) |
    ------        ------------------------------
    | 3 |         | C(3,1) | C(3,2) | C(3,3) |
    ------        ------------------------------
```

In the example above, array C is two dimensional. There are three instances of B in data group A and each instance of B contains a number of elements of array C. Correspondingly the range of the first dimension of array C is a constant three and the range of the second dimension which may depend on the subscript value of the first dimension is specified in vector SIZE.C. SIZE.C(1) equals to four implies that there are four elements of array C in the first instance of B, the value of SIZE.C(2) specifies the number of elements of array C in the second instance of B, and so on.

<u>END.X</u>

If X is a repeating member of a data structure, END.X can be used to specify the range of the rightmost dimension of array X as alternative to the use of SIZE.X.

END.X is a boolean array. If X(I1,I2,...,In) is an n dimensional array, then the associated control array END.X(I1,I2,...,In) is an n dimensional array, too. The range of array dimensions of END.X are the same as the corresponding array dimensions of X. The value of END.X determines the range of the rightmost dimension of array X in the following way. For every n-1 tuple (I1,I2,...,In-1) which is a possible combination of the leftmost n-1 subscripts of array X, there exists a sequence of elements in END.X array with the same left n-1 subscript values, i.e. {END.X(I1,...,In-1,In)| 1<=In}. If END.X(I1,...,In-1,m) is a boolean true and all the elements of {END.X(I1,...,In-1,In)| 1<=In<m} are false, then there are exactly m elements in array X with (I1,...,In-1) as their leftmost n-1 subscripts. The values in END.X may depend on the values in array X, i.e. the number of repetition may depend on the data in X.

Example:

For the same array C mentioned above, we may use a two dimensional control array END.C to specify the range of the second dimension of array C as follows.

```
    A IS GROUP (B(3)) ;
    B IS GROUP (C(*)) ;
    C IS FIELD;
    END.C(SUB1,SUB2) = IF SUB1=1 THEN (SUB2=4)
                       ELSE IF SUB1=2 THEN (SUB2=2)
                       ELSE IF SUB1=3 THEN (SUB2=3) ;
            C
            ----------------------------------------
            | C(1,1) | C(1,2) | C(1,3) | C(1,4) |
            ----------------------------------------
            | C(2,1) | C(2,2) |
            ------------------------------
            | C(3,1) | C(3,2) | C(3,3) |
            ------------------------------

        END.C
            ----------------------------------------
            |   F    |   F    |   F    |   T    |
            ----------------------------------------
            |   F    |   T    |
            ------------------------------
            |   F    |   F    |   T    |
            ------------------------------
```

In the first row of END.C the first boolean true comes in the fourth element, therefore, the fourth element is the last element in the first row of array C. Similarly, the second element of the second row of END.C is true implies that there are only two elements in the second row of array C.

Example:

We will show how the END control variable can be used to specify a varying number of repetitions by finding the greatest common divisor of two positive integers M and N. Euclid's algorithm is used here.

```
MODULE: TEST ;
SOURCE: IN ;
TARGET: OUT ;

IN IS FILE (INR) ;
    INR IS REC(M,N) ;

OUT IS FILE (OUTR) ;
    OUTR IS REC(GCD) ;
```

```
WK IS GROUP (WKG(*)) ;
  WKG IS GROUP (WK1,WK2) ;
(M,N,GCD,WK1,WK2) IS FIELD NUM(4) ;

WK1(SUB1) = IF SUB1=1 THEN M
            ELSE IF WK1(SUB1-1)>WK2(SUB1-1) THEN
                    WK1(SUB1-1)-WK2(SUB1-1)
            ELSE WK2(SUB1-1) ;

WK2(SUB1) = IF SUB1=1 THEN N
            ELSE IF WK1(SUB1-1)>WK2(SUB1-1) THEN
                    WK2(SUB1-1)
            ELSE WK1(SUB1-1) ;

END.WKG(SUB1) = WK1(SUB1)=WK2(SUB1) ;

IF END.WKG(SUB1) THEN GCD = WK1(SUB1) ;
```

POINTER.X

If X is a record of a keyed input file F, the instances of the record X can be selected and ordered according to the value of a control variable POINTER.X. The control variable POINTER.X has the same number of dimensions and the same shape as the array X. For every value in the control variable POINTER.X, a record instance in the file F with that key value will be presented in the corresponding element of array X. In order to use POINTER control variable for selecting and ordering the records in a keyed file, one of the field in records should be declared as a key in the file declaration statement. The content of the POINTER control variable is used as the key to access the corresponding record from the keyed file.

A keyed file may either have sequential or index sequential organization. If the file is index sequential, the records stored in the file may be in any order. However, if the file is actually a sequential file, then the records have to be sorted in an ascending order according to the key field and the keys used to access the records should also be sorted in the same order. This is an implementation restriction. Without this restriction we can not read all the records we want from that file in one pass.

When a keyed file is declared as a source and a target file, the target file will be an updated version of the source file. Effectively only the records being selected may be modified. For the rest of the file they are kept intact in the target file. This mechanism makes the update of sequential or index sequential file much easier to specify. Since a key value may occur more than once in the POINTER array, the corresponding (one) record will be accessed, possibly updated, and written out several times. In order to make sure every update to the same record is effective, the updates have to be done sequentially. We can envisage that a new version of the keyed file is created after one record is updated and every update is done on the most recent version of the file.

Example:

In the following MODEL specification a source file INVEN is declared as a keyed file. STOCK$ in the record INVREC is the key field of INVEN file. Since the control variable POINTER.INVREC is equal to the field STK in file TRAN, the INVREC records will be ordered according to the values in the STK field.

```
MODULE: MINSALE ;
SOURCE: TRAN, INVEN ;
TRAN IS FILE (SALEREC(*)) ;
    SALEREC IS RECORD (CUST$,STK,QUANTITY) ;
        CUST$ IS FIELD(CHAR(5)) ;
        STK IS FIELD(CHAR(8)) ;
        QUANTITY IS FIELD(CHAR(3)) ;

INVEN IS FILE (INVREC(*))
  KEY STOCK$
  ORG ISAM ;
    INVREC IS RECORD(STOCK$,SALPRICE,QOH) ;
        STOCK$ IS FIELD(CHAR(8)) ;
        SALPRICE IS FIELD(NUMERIC(5)) ;
        QOH IS FIELD(NUMERIC(5)) ;

POINTER.INVREC = TRAN.STK ;
```

<u>FOUND.X</u>

If X is a record in a keyed file, then it is accessed through the value of a POINTER control variable. It may happen that the key value used to access the record does not match with any record. The accessing would fail. The user may test the value in a control variable called FOUND.X to find out whether a record with some specific key exists or not. This informaton may be used to decide whether a new record should be added into the file or an old record should be updated. The control variable FOUND.X has the same shape as array X and POINTER.X. Its data type is boolean.

<u>LEN.X</u>

If X is a field in some record and its data type is variable length character string, then the actual length of X is specified by the control variable LEN.X which is used to disassemble the input or output records. Corresponding to every element of array X, there is an element in LEN.X. The values in the array LEN.X are integers. We can use any integer type expression to define LEN.X. The only restriction is that the content of LEN.X should not depend upon any data physically positioned in a record after the data field X.

<u>NEXT.X</u>

If X is a field in an input sequential file, the control variable NEXT.X can be used to denote the same field in the next physical record of the file. Although the next record usually means the record with a subscript value one larger than the current record, it may not be true when the current record is the last record in some group. The problem is caused by the fact that the user is dealing with structured data but the real data in the external file is in a linear form. Sometimes the information used to transform a sequence of records into a structured form can only be conveniently expressed in the way that the records are physically contiguous. For example, we may want to compare the value of a key field in two adjacent records to determine whether a record is the last record in a group or not. The fact that the current record and the next record may or may not be in the same group causes trouble in referencing the next record.

Example:

Suppose the records in a transaction file contain a customer number and some relevant information and the records are sorted according to the value of the customer number field. We may use the following specification to describe the data structure.

```
TRANSACTION IS FILE (CUSTOMER(*)) ;
    CUSTOMER IS GROUP (TRANS_REC(*)) ;
        TRANS_REC IS RECORD (CUSTOM_NO,INFORMATION) ;
            CUSTOMER_NO IS FIELD (PIC'99999999') ;
I IS SUBSCRIPT ;
J IS SUBSCRIPT ;
END.TRANS_REC(I,J) =
          CUSTOMER_NO(I,J)^=NEXT.CUSTOMER_NO(I,J) ;
```

The term NEXT.CUSTOMER_NO(I,J) in the last assertion can not be replaced by CUSTOMER_NO(I,J+1) because there may not be a record with this pair of subscript values. The restriction in using the control variable NEXT.X is that the position of X field in a record should be fixed, i.e. the fields to the left of the field X can not be variable length strings or repeating with a variable number of times. Otherwise, the field X in the next record may not be located correctly.

<u>SUBSET.X</u>

If X is a record in an output file, then the control variable SUBSET.X can be used to selectively omit some records from an output file. The SUBSET.X control variable is a boolean array of the same shape as the array X. When an element in the SUBSET.X has a value of boolean true, the corresponding record X will be put into the output file. On the other hand, if the element has a value of boolean false, the corresponding record will not be put into the output file. It should be noted that the use of SUBSET control variable does not affect any other computations. Only a subset of records X may be omitted from the output file.

CHAPTER 3

SYNTAX ANALYSIS PROGRAM

The first phase of the MODEL processor analyzes the syntax and other local semantics of individual statements. Advanced state-of-the-art syntax analysis techniques are used here which have proved to be invaluable. Specifically, the capability to generate the parser automatically has enabled rapid development changes. In addition to checking the MODEL statements for syntactic and some semantic errors, this phase also stores the statements in an internal associative form for later processing.

3.1 EBNF, SAPG, AND THE SAP

3.1.1 SPECIFICATION OF MODEL USING EBNF AND THE SAPG

The syntax Analysis Program (SAP) for the MODEL statements is generated automatically by a Syntax Analysis Program Generator (SAPG). As shown in Figure 3.1, the SAPG produces the Syntax Analysis Program (SAP) for analyzing MODEL statements, based on a specification of the MODEL language expressed in the EBNF/WSC (extended Backus Normal Form With Subroutine Calls) meta language.

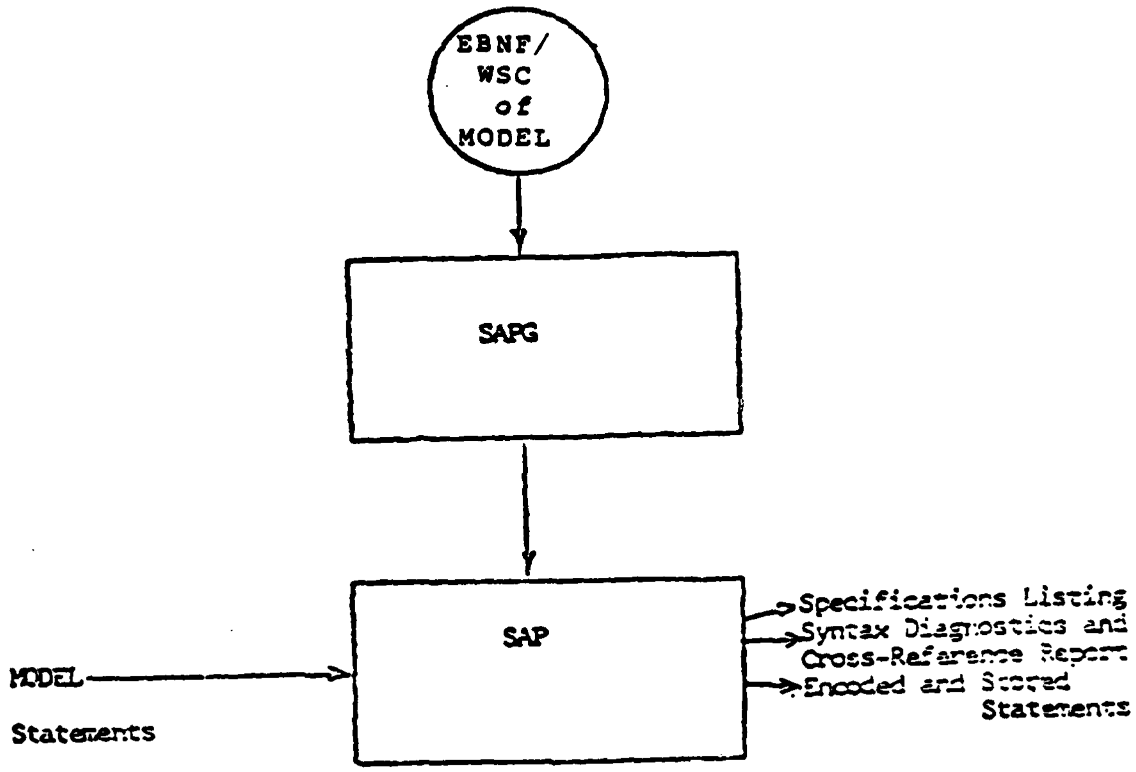

EBNF/ WSC of MODEL
SAPG
SAP
MODEL Statements
Specifications Listing
Syntax Diagnostics and Cross-Reference Report
Encoded and Stored Statements

Figure 3.1 Block Diagram of SAPG and SAP

The EBNF/WSC includes the traditional concepts of BNF. BNF uses sequences of characters enclosed in angle-brackets < > called non-terminals to give names to grammatical units, for which substitutions may be made. It also uses sequences of characters not enclosed in brackets which are in the object language (in this case MODEL). BNF consists of a series of production rules or substitution rules of the form "A::=B" where "A" is a single non-terminal symbol and "B" is one or more alternative sequences of terminal or non-terminal symbols that can be substituted for A. The alternatives are separated by the meta-symbol "|". To facilitate language description, BNF was extended to EBNF with two more well-known meta-symbols: [ ] representing optionality and [ ]* representing zero or more repetitions.

The specification of MODEL that is input to the SAPG consists not only of the syntax specification of MODEL, but also of subroutine names embedded within the EBNF; therefore the name "EBNF With Subroutine Calls" (EBNF/WSC). The SAPG provides a capability to branch to these subroutines upon successful recognition of a syntactic unit. Thus, they can complete the SAP to enable it to check some of the statement semantics, to encode, to produce error messages, and to store the MODEL statements for later retrieval. The invocations of these subroutines themselves are written manually. The definition of the MODEL language in EBNF/WSC appears in Figure 3.2. The subroutines to be invoked are indicated between slaches (/.../). Note that subroutine calls are made after the successful recognition of syntactic units up to that point.

The SAP generated by the SAPG according to the EBNF/WSC is supplemented and linked with the routines. The SAP accepts statements in MODEL and checks them for syntactic correctness, and local semantics. It produces a listing of the statements, syntax diagnostics, an encoded stored version of the MODEL statements, syntax trees for the assertions and a cross-reference report.

```
<MODEL_SPECIFICATION>::=[<MODEL_BODY_STMTS> /CLRERRF/ ]*
                         /STMT_FL/  <MODEL_SPECIFICATION>
<MODEL_BODY_STMTS>::=    /E(80)/
                        MODULE <MODULE_NAME_STMT>
                        |SOURCE <SOURCE_FILES_STMT>
                        |TARGET <TARGET_FILES_STMT>
                        | @ _END @ /ENDINP/
                        | <DCL_DESCRIPTION>
                        | <BLOCK_BEGIN>
                        | <BLOCK_END>
                        | <OLD_FILE_STMT>
                        | /ASSINIT/ <ASSERTIONS> /STRHS/
<DCL_DESCRIPTION> ::= 1 /INTDCL/ /INTMVAR/ /MEMINIT/ /SVMEM/
                        <DATA_SPEC>
                        [, /E(108)/ <INTEGER> /CRDCL/
                         /INTMVAR/ /MEMINIT/ /SVMEM/
                         <DATA_SPEC> ]*     /STDCL/ <ENDCHAR>
<DATA_SPEC>    ::= <DCL_MVAR> [( <OCCSPEC> )] [ <IS> ]
                   <ATTR_SPEC> /SVDCL/
<ATTR_SPEC>    ::= <FILE> /SVF/ /SVFLNM/ <FILE_DESC>
                   <STORAGE_DESC>          /STDEV/
                 | <RECORD>                /SVR/
                 | <FIELD_STMT> /STDFLD/ /SVD/
                 | [<GROUP>]               /SVG/
<BLOCK_BEGIN> ::= BLOCK /BLKINIT/ [ <NAME> /SVLBL/ ] /E(2)/
                  : [ <BLOCK_SPEC> ]* /SVBLOK/  <ENDCHAR>
<BLOCK_SPEC>   ::= <SOLUTION> | <ITERATION> | <REL_ERROR>
<SOLUTION>     ::= [ SOLUTION ] METHOD [ <IS> ] /E(62)/
                   <METHODS> /SVMETH/ [ , ]
<METHODS>      ::=  NEWTON | GAUSS_SEIDEL | G_S | JACOBI
<ITERATION>    ::=  [ <MAXIMUM> ] <ITER> [ <IS> ] /E(4)/
                    <NUMBER> /SVITER/ [ , ]
<MAXIMUM>      ::=  MAX | MAXIMUM
<ITER>         ::=  ITER | ITERATION | ITERATIONS
<REL_ERROR>    ::= [ RELATIVE ] <ERROR> [ <IS> ] /E(5)/
                   <NUMBER> /SVERR/ [ , ]
<ERROR>        ::=  ERR |  ERROR
<BLOCK_END> ::= <END> /BLKEND/ [ <NAME> /CHKLBL/ ] <ENDCHAR>
<END>         ::= /ENDID/
<ASSERTIONS>::=/E(14)/<CONDITIONAL> |
               /SVASSR/ /INTMVAR/ <MVAR> /STMVAR/ /SVCMP1/
               [<IS>/SVNXOP/]<DDL_OR_RHS>
<CONDITIONAL>::=IF /SVAAS1/ /SVOP1/ /SETBIT/ /E(18)/
               <BOOLEAN_EXPRESSION> /SVCMP1/ /E(38)/
               THEN /SVNXOP/ <SIMPLE_ASSERTION> /SVNXCMP/
               [ELSE /SVNXOP/ <ASSERTION> /SVNXCMP/] /STALL/
<ASSERTION>::= /E(14)/ <CONDITIONAL> | <SIMPLE_ASSERTION>
```

Figure 3.2 Definition of MODEL language in EBNF/WSC

```
<DDL_OR_RHS>::=/INTODDL/ <DATA_DESC_STMT> /FREETMP/
               | /E(33)/ <INTOAS> <ASSERTION_BRANCH>
                 <ENDCHAR>
<ASSERTION_BRANCH>::= <DEF_EXPRESSION>
                    | <BOOLEAN_EXPRESSION>/SVNXCMP/ /STALL/
<DEF_EXPRESSION>::=  /INTSUB/ ( <VALUE_LIST> ) /FREESUB/
<VALUE_LIST>::= ( /CRSUB/ /DECPP/ <VALUE_LIST>
                  [, <VALUE_LIST> ]* ) /INCPP/
               | [<SIGN> /SVOPP/] <NUMBER> /STNUM/ /STASS/
<INTOAS>::=/INTOASS/
<SIMPLE_ASSERTION>::= /SVASAE1/ /INTMVAR/ <MVAR> /STMVAR/
                      /SVCMP1/ /E(23)/ = /SVNXOP/
                      <BOOLEAN_EXPRESSION> /SVNXCMP/ /STALL/
                      <ENDCHAR>
<SUB_VARIABLE>::= /SETSUBV/ <VAR> /SVCMP1/
          [(/SVNXOP/ /SETBIT/ /E(22)/
           <BOOLEAN_EXPRESSION> /SVNXCMP/ [,/SVNXOP/
           <BOOLEAN_EXPRESSION>/SVNXCMP/]*
          /E(24)/ ) ] /STALL/
<BOOLEAN_EXPRESSION>::= /E(82)/ /SVBEXP/ <COND_EXP>
                      | <BOOLEAN_TERM> /SVCMP1/
                        [<OR> /SVNXOP/ <BOOLEAN_TERM>
                          /SVNXCMP/]*         /STALL/
<COND_EXP>::= IF /SVCOND/ /E(3)/ <BOOLEAN_EXPRESSION>
              /SVCMP1/ /E(79)/ THEN /SVNXOP/
              <BOOLEAN_EXPRESSION> /SVNXCMP/ /E(12)/ ELSE
              /SVNXOP/ <BOOLEAN_EXPRESSION> /SVNXCMP/
              /STALL/
<OR>::= /OR_REC/
<BOOLEAN_TERM>::= /E(83)/ /SVBT1/ <BOOLEAN_FACTOR> /SVCMP1/
                  [_/SVNXOP/ <BOOLEAN_FACTOR> /SVNXCMP/]*
                  /STALL/
<BOOLEAN_FACTOR>::= /E(82)/ /SVBF1/ <CONCATENATION> /SVCMP1/
                    [<RELATION> /SVNXOP/ <CONCATENATION>
                    /SVNXCMP/]* /STALL/
<RELATION>::= /RELREC/
<CONCATENATION>::= /E(84)/ /SVCON/ <ARITH_EXP> /SVCMP1/
                   [ <CONCAT> /SVNXOP/ <ARITH_EXP>
                    /SVNXCMP/]* /STALL/
<CONCAT>::= /CATREC/
<ARITH_EXP>::= /E(81)/ /SVAE/ [<SIGN> /SVOP1/]
               <TERM> /SVCMP1/ [<OPS> /SVNXOP/ <TERM>
                                 /SVNXCMP/]* /STALL/
<TERM>::= /E(87)/ /SVTERM/ <FACTOR> /SVCMP1/
          [<MOPS> /SVNXOP/ <FACTOR> /SVNXCMP/]* /STALL/
<FACTOR>::= /E(85)/ /SVFAC/ [ /SVOP1/] <PRIMARY> /SVCMP1/
            [<EXPON> /SVNXOP/ <PRIMARY> /SVNXCMP/]* /STALL/
<EXPON>::= /EXPREC/
```

Figure 3.2 Definition of MODEL language in EBNF/WSC

```
<PRIMARY>::= /E(86)/ /SVPRIM/ <IS_PRIM> /SVCMP1/ /STALL/
<IS_PRIM>::= ( <BOOLEAN_EXPRESSION> /E(24)/ )
             | <NUMBER> /STNUM/ | <STRING_FORM>
             | <FUNCTION_CALL> | <SUB_VARIABLE>
<STRING_FORM>::= ' /SETSTRN/ [ <STRING>  /SVSTRNG/] /E(26)/
                 ' /ADLEX/ [B /STBIT/ /E(1)/ <B_SUFX>]
                 /STNUM/
<FUNCTION_CALL>::= <FUNCTION_NAME> /STFUN/
                   /SETFUNC/ [(/SVNXOP/ <BOOLEAN_EXPRESSION>
                   /SVNXCMP/ [,/SVNXOP/ <BOOLEAN_EXPRESSION>
                   /SVNXCMP/ ]* ) ] /STALL/
<FUNCTION_NAME>::= /FNCHECK/
<MVAR>::=  ( <SUB_VARIABLE> /SVMVAR/
            [, <SUB_VARIABLE> /SVMVAR/ ]* )
           | <SUB_VARIABLE> /SVMVAR/
<VAR>::= /SETVAR/ /INITQNM/ /E(68)/ <NAME> /ADLEX/ /MKQNM/
         [. /ADLEX/ /E(68)/ <NAME> /ADLEX/ /MKQNM/]*
         /STR_CON/
<DCL_MVAR> ::= ( <VAR> /SVMVAR/ [, <VAR> /SVMVAR/ ]* )
               | <VAR> /SVMVAR/
<B_SUFX>::= /BITSTR/
<QNAME>::= /INITQNM/ /E(68)/ <NAME>  /MKQNM/
    [ .  /E(68)/ <NAME>  /MKQNM/ ] *
<STRING>::= <STRING_CONST>
<OPS>::= /OPREC/
<MOPS>::= /MOPREC/
<TEST>::= /TESTBIT/
<MODULE_NAME_STMT>::= /E(63)/: /E(64)/ <NAME> /STMOD/
                      <ENDCHAR>
<SOURCE_FILES_STMT>::= [<FILE_KEYWORD>] /E(75)/ /INITSFL/ :
                       <SOURCE_FILELIST> /STSRC/ <ENDCHAR>
<FILE_KEYWORD>::= FILES|FILE
<SOURCE_FILELIST>::= /E(76)/ <NAME> /SVSRC/
                     [, /E(76)/ <NAME> /SVSRC/]*
<TARGET_FILES_STMT>::= [<FILE_KEYWORD>] /E(77)/ /INITTFL/ :
                       <TARGET_FILELIST> /STTAR/ <ENDCHAR>
<TARGET_FILELIST>::= /E(78)/ <NAME> /SVTAR/
    [, /E(78)/ <NAME> /SVTAR/ ]*
<DATA_DESC_STMT>::=  <DATA_DESCRIPTION> <ENDCHAR>
<DATA_DESCRIPTION>::=
  <FILE_STMT>    /STFILE/
  |<RECORD_STMT> /STREC/
  |<GROUP_STMT>  /STGRP/
  |<FIELD_STMT> /STFLD/
  |<SUB_STMT>    /STSUBST/
<SUB_STMT>::=<SUBSCRIPT>/MEMINIT/ /SVMEM/ [( <OCCSPEC> )]
<SUBSCRIPT>::= SUB | SUBSCRIPT | SUBSCRIPTS
<FILE>::= FILE | REPORT | FILES | REPORTS
```

Figure 3.2 Definition of MODEL language in EBNF/WSC

```
<RECORD_STMT>::= <RECORD> /MEMINIT/ [(] <ITEM_LIST> [)]
<RECORD>  ::=  REC | RECORD | RECORDS
<ITEM_LIST>::= /E(52)/<ITEM> [[,] <ITEM>]*
<ITEM>::= <NAME> /SVMEM / [ . <NAME> /SVMEM/ ]*
          [( <OCCSPEC> )]
<OCCSPEC>::=  <STAR> /SVSTAR/ | <MINOCC>/SVMNOC/ [<MAXOCC>]
<STAR>::= /STARREC/
<MINOCC>::=<INTEGER>
<MAXOCC>  ::=  [:/E(51)/]<INTEGER> /SVMXOC/ /CKMNMX/
               |            <INTEGER> /SVMXOC/ /CKMNMX/
<GROUP_STMT>::= <GROUP>/MEMINIT/ [(] <ITEM_LIST> [)]
<GROUP>  ::=  GRP | GROUP | GROUPS
<FIELD_STMT>::=  <FIELD> /SVFLD/ <FIELD_ATTR>
<FIELD>  ::=  FLD | FIELD | FIELDS
<FIELD_ATTR>::= [(]  <TYPE> /SVFDTP2/[ <LENG_SPEC>]
                [,]  [<LINE_SPEC>]  [,]  [<COL_SPEC>]  [)]
<LENG_SPEC> ::= ( /E(48)/ <MIN_LENGTH> [ <MAX_LENGTH> ]
                 /E(49)/ )
                |<MIN_LENGTH> [<MAX_LENGTH>]
<MIN_LENGTH>::= <INTEGER> /SVMNFLN/
<LINE_SPEC>::= LINE /E(53)/ /E(54)/ /E(55)/
               (<INTEGER> /SVLINE/)
<COL_SPEC>::= COL /E(90)/ /E(91)/ /E(92)/
              ( <INTEGER> /SVCOL/ )
<TYPE>::= /E(47)/ <PIC_DESC> | <STRING_SPEC>  | <NUM_SPEC>
<PIC_DESC>::= <PIC_TYPE> /E(67)/ /SVPIC/
              ' [ <STRING> /SVPICST/ ] ' /STPIC/
<PIC_TYPE>::= PIC | PICTURE
<STRING_SPEC>::= <STRING_TYPE> /SVSTRTP/
<STRING_TYPE>::= CHAR | CHARACTER | BIT | NUM | NUMERIC
<NUM_SPEC>::= <NUM_TYPE> /SVNUMTP/ [ <FIXFLT> /SVMOD/ ]
<NUM_TYPE>::= BIN | BINARY | DEC | DECIMAL
<FIXFLT>::= FIX | FIXED | FL | FLOAT | FLT
<MAX_LENGTH>::= [:] <INTEGER> /SVMXFLN/
                | , /E(46)/ <SINTGR> /SVSCALE/
                | <INTEGER> /SVMXFLN/
<SINTGR>::= - /E(50)/ <INTEGER> /NEGATE/ | <INTEGER>
<NUMBER> ::= /SETNUM/ <INITNUM> /E(65)/ <RECNUM>
<RECNUM>::= /RECNUM/
<INITNUM>::= /INITNUM/
<SIGN>::= + | -
<RECG>::= <RECORD> | <GROUP>
<KEY>::=KEY|SEQUENCE
<CODE>::=EBCDIC|BCD|ASCII
<ANY>::= <NAME>|<INTEGER>
<NO_TRKS>::= 7|9
<DENSITY>::= 200|556|800|1600|6250
<PARITY>::= ODD|EVEN
```

Figure 3.2 Definition of MODEL language in EBNF/WSC

```
<TYPEDSK>::= 2314|2311|3330|2305 | 3330-1
<ORG>::=ORG|ORGANIZATION
<ORG_TYPE>::= /E(7)/ISAM|SEQUENTIAL|SAM|INDEXED_SEQUENTIAL
<ENDCHAR>::= /E(74)/ <END_CHAR> /STMTINC/
<END_CHAR>::= /SVENDC/
<STRING_CONST>::=/CHARSTR/
<NAME>::=/NAMEREC/
<INTEGER>::=/INTREC/
<IS>::= IS | = | ARE
<FILE_STMT>::=  <FILE> /SVFLNM/ /MEMINIT/ <SON_DESC>
                <FILE_DESC> <STORAGE_DESC> /STDEV/
<SON_DESC>::=( <ITEM_LIST> )
             | <RECG> [NAME] [<IS>] [(] <ITEM> [)]
<OLD_FILE_STMT>::= <FILE> [NAME] [<IS>] /E(56)/ /MEMINIT/
                      /INTMVAR/
                  <DCL_MVAR> /SVFLNM/
                  <RECG> [NAME] [<IS>] [(] <ITEM> [)]
                  <FILE_DESC> /STFILE/
                  <STORAGE_DESC> /STDEV/  <ENDCHAR>
<FILE_DESC>::= [ STORAGE [NAME] [<IS>] /E(44)/ <NAME>
                  /SVSTNM/]
              [<KEY> [NAME] [<IS>] /E(45)/ <NAME> /SVKEY/]
              [<ORG> [<IS>] <ORG_TYPE> /SVORG3/]
<STORAGE_DESC>  ::= [DEVICE [<IS>] <DEVICE>] /SVDEV/
          [RECORD /E(57)/][FORMAT [<IS>] <REC_FMT>]/SVRECF/
          <BLK_REC_VOL>
          [<TAPE_DESC>] [<DISK_DESC>]
          [HARDWARE]  [SOFTWARE]
<DEVICE>  ::= /E(61)/ TAPE | DISK/SETDEVB/
      | CARD /SETDEVC/     | PRINTER /SETDEVP/
      | PUNCH /SETDEVU/    | TERMINAL /SETDEVT/
<REC_FMT>  ::= /E(69)/ FIXED|VARIABLE|VAR_SPANNED|UNDEFINED
<BLK_REC_VOL>  ::=
                  [ [MAX] /E(70)/ /E(71)/ BLOCKSIZE [<IS>]
                       <INTEGER> /SVBLK/ ]
                  [ [MAX/E(59)/] RECORDSIZE [<IS>] /E(72)/
                       <INTEGER>/SVRCSZ/]
                  [ VOLUME [NAME] [<IS>] /E(60)/ <NAME>
                       /SVVOL/ [,/E(60)/<NAME>]* ]
<TAPE_DESC>  ::= [<TRACKS> [<IS>] /E(66)/<NO_TRKS>/SVTRK2/ ]
                [PARITY [<IS>] /E(66)/ <PARITY>/SVPAR2/]
                [DENSITY [<IS>] /E(66)/ <DENSITY> /SVDEN2/]
                [ [TAPE] LABEL [<IS>] <LABEL_TYPE>/SVLAB2/]
                [START [FILE] [<IS>] /E(66)/ <INTEGER>
                     /SVSTFL2/]
                [[CHAR] CODE [<IS>] <CODE> /SVCC/ ]
<TRACKS>  ::=  NO_TRKS | TRACKS
<LABEL_TYPE>  ::= /E(58)/ IBM_STD|ANSI_STD|NONE|BYPASS
```

Figure 3.2 Definition of MODEL language in EBNF/WSC

```
<DISK_DESC>  ::=  [UNIT [<IS>] /E(9)/ <TYPEDSK> /SVUNIT2/]
                   [<CYLINDERS>/SVUCYL/ [<IS>] /E(66)/
                        <INTEGER> /SVQTY2/]
<CYLINDERS>  ::=  NO_CYLS | CYLINDERS
<HARDWARE>::= [[COMPUTER] MODEL [<IS>] <ANY>
<SOFTWARE>::= [[OPERATING] SYSTEM [<IS>] <ANY>]
```

Figure 3.2 Definition of MODEL language in EBNF/WSC

3.1.2 HOW THE SAPG PRODUCES THE SAP

The SAPG is a parser generator. It accepts a specification in the language EBNF/WSC and produces a parser program (SAP). It performs this in three passes over the set of productions.

In pass 1, each production is scanned, and its components are encoded into a set of tables. Non-terminal symbols appearing on the left-hand-side of a production (new production names) are put into a symbol table (LHS-NT-SYM-TAB), while non-terminals appearing on the right-hand-side of a production are put into another symbol table (RHS-NT-SYM-TAB). Terminal symbols in a production are put into a terminal symbol table (TERM-SYM-TAB). Subroutine calls are put into yet another table (SUB-TAB).

In pass 2, the symbolic references in RHS-NT-SYM-TAB (i.e. non-terminals on the right-hand-side of the original production) are resolved. Pass 2 checks that each non-terminal symbol in RHS-NT-SYM-TAB is defined, and links it to the corresponding entry in LHS-NT-SYM-TAB. Undefined non-terminals as well as circularly-defined non-terminals can be detected in these table searches.

Pass 3 of the SAPG is the code-generation phase that produces the SAP in PL/I. It is only entered if no errors were encountered in the previous phases. For each EBNF/WSC production, a PL/I procedure is generated. Each one returns a bit: 1 if the recognition was successful; 0 if it was unsuccessful. The exclusive nature of EBNF production rules and alternatives is effected by generating nested PL/I IF-THEN-ELSE statements. Repetition zero or more times is effected by generating a GO TO to the statement testing for recognition. Subroutine names embedded in the EBNF/WSC get a CALL generated for them in place. Calls to other subroutines not explicit in the EBNF/WSC are also generated. These include "housekeeping" subroutines of the SAP and calls to LEX, a subroutine to scan and return the next token in the object language.

To illustrate the code that the SAPG generates, consider the following representative production rule in the EBNF/WSC and the PL/I code that corresponds:

```
    <FIELD_STMT>::= <FIELD> /SVFLD/ <FIELD_ATTR> /STFLD/
```

The PL/I code that is generated for it by the third pass of the SAPG would be the following:

```
FIELD_STMT: PROCEDURE RETURNS(BIT(1));
CALL $MARK;
IF FIELD( ) THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
CALL SVFLD;
IF FIELD_ATTR( ) THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
CALL STFLD;
CALL $SUCCES; RETURN('1'B);
END; ELSE DO; CALL $SUCCES; RETURN('1'B); END;
END; ELSE DO; CALL $FAIL; RETURN('0'B); END;
END FIELD_STMT;
```

The above code generated by the SAPG would become one procedure in the SAP. Note that the name that the language definer uses in the production rule are preserved in the generated SAP code. The subroutines beginning with dollar signs ($) are "housekeeping" routines that are internal to the mechanisms of SAPG-generated code.

3.2 SUPPORTING SUBROUTINES FOR EBNF OF MODEL

A refined system flowchart of the SAPG and SAP showing the types of supporting routines appears in Figure 3.3.

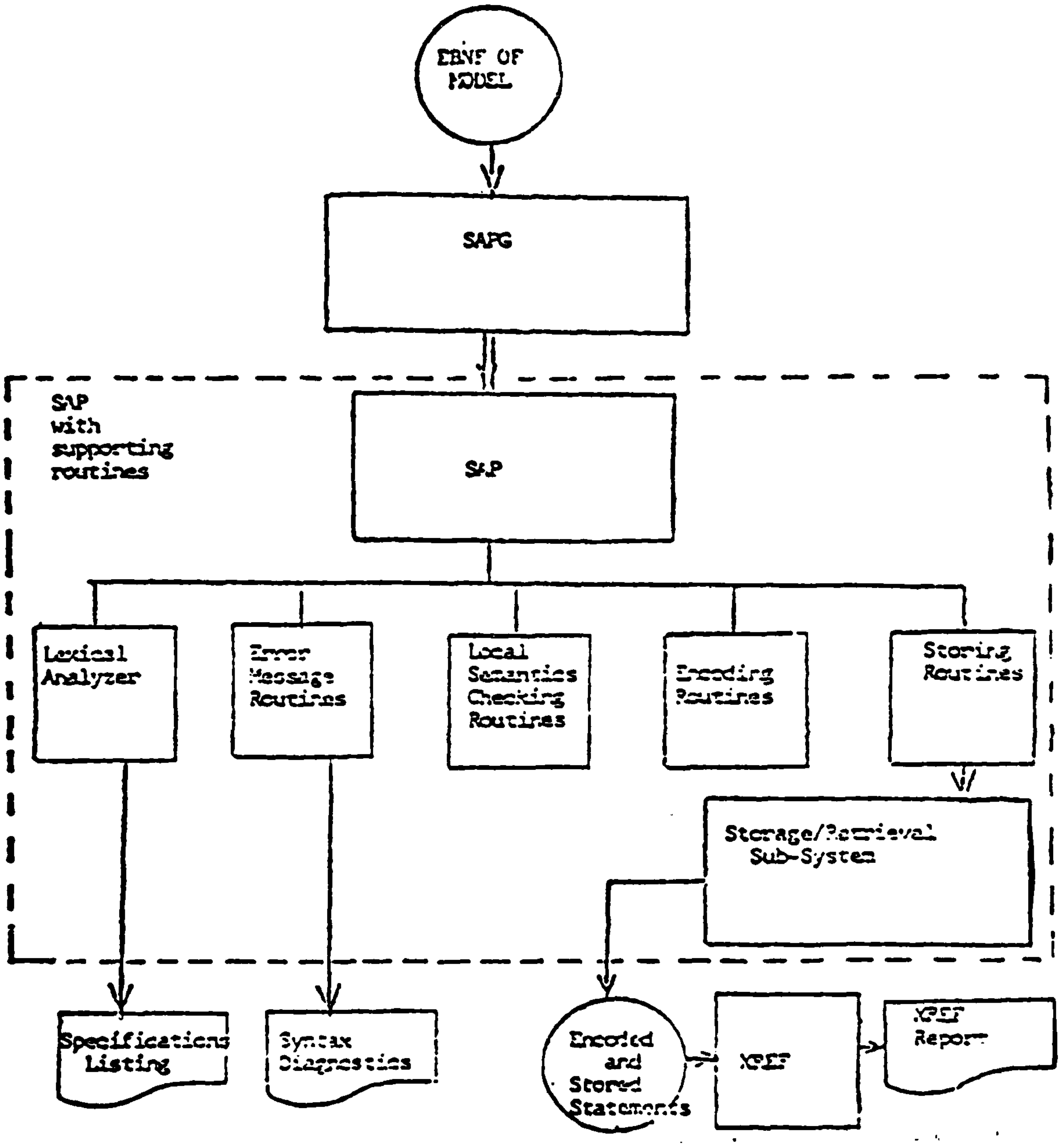


Fig. 3.3 More Detailed View Of SAPG and SAP With Supporting Subroutines

The manually-written syntactical supporting routines are of one of several types:

(1) a lexical analyzer which returns tokens of syntactic units to the SAP for analysis;

(2) statement semantics checking routines;

(3) error message handling routines;
(4) encoding routines to compact information for further efficient processing; and
(5) statement storage routines.

The cross-reference report produced during this phase is generated by a manually-written program (XREF) and is described in section 3.4.

A discussion on how to decide where to insert subroutines as well as a tabular summary of all routines used appears in section 3.2.

3.2.1 THE LEXICAL ANALYZER

The purpose of the lexical analyzer is to scan for syntactic units or "tokens", using such delimiters as blanks and certain punctuation marks, and to return tokens to the Syntax Analysis Program (SAP) for syntactic checking. The automatically-generated SAP calls upon the lexical analyzer (LEX) whenever it needs the next token. The lexical analyzer is based on the finite state machine concept. Each state of the machine corresponds to a condition in the lexical processing of a character string. At each state, a character is read, an action is taken based on the character read (such as concatenating the current character to previous ones or returning the entire token to the SAP), and the machine changes to a new state. The character classes for the MODEL language, for the purposes of lexical analysis, appear in Table 3.1. These classes divide the entire character set into categories such as illegal characters, delimiters, "normal" characters, ... etc. The state transition matrix for the MODEL language appears in Table 3.2. The rows of the matrix represent the character classes of the previous character, while the columns represent those of the current character. The entries in the matrix indicate the action to be taken and the next state. The action taken in each state is summarized in Table 3.3. The actions involve such steps as concatenating of a character, ignoring a character, detecting an illegal character, returning a complete token to the SAP, ... etc., and setting a "next state".

Class	Character Set	Explanation
0	A B ... Y Z _ # @	Characters in names
1	space	Delimiter
2	0 1 2 ... 9	Numerals
3	. (+ _) ; , % : ' "	Delimeters
4		
5	<	Delimeter in logical exp
6	|	"OR" symbol
7	*	Multi. or comment in "/*"
8		"NOT" symbol
9	–	minus symbol
10	/	Division or comment
11	>	Delimeter in logical exp
12	=	Delimeter and logical exp
13	all others	Illegal

Table 3.1 Character Classes for MODEL Language

Character Class (next)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
(current)														
0	1	2	1	2	2	2	2	2	2	2	2	2	2	7
1	1	3	1	5	1	1	1	1	1	1	1	1	1	7
2	1	2	1	2	1	2	2	2	2	2	2	2	2	7
3	2	2	2	2	2	2	2	2	2	2	2	2	2	7
4	2	2	1	2	2	2	2	2	2	2	2	2	2	7
5	2	2	2	2	2	2	2	2	2	2	2	2	1	7
6	2	2	2	2	2	2	1	2	2	2	2	2	2	7
7	2	2	2	2	2	2	2	1	2	2	2	2	2	7
8	2	2	2	2	2	1	2	2	2	2	2	1	1	7
9	2	2	2	2	2	2	2	2	2	2	2	1	2	7
10	2	2	2	2	2	2	2	6	2	2	2	2	2	7
11	2	2	2	2	2	2	2	2	2	2	2	2	1	7
12	2	2	2	2	2	2	2	2	2	2	2	2	2	7
13	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Table 3.2 State Transition Matrix for MODEL Lexical Analyzer

Action 1:	Concatenate next character to current token
Action 2:	End word with next character
Action 3:	Skips blanks sequence
Action 4:	Reserved (not used)
Action 5:	Scan forward one character and save as token
Action 6:	Comment bracket; Scan to end of comment
Action 7:	Illegal character(s); print error message

Table 3.3 Lexical Analysis Actions

3.2.2 STATEMENT SEMANTICS ANALYSIS

Some of the semantics of the specification statements can be checked during the syntax analysis phase. Such routines can check that a range or condition on a syntactic unit is locally correct. These routines do not and cannot check the overall consistency, completeness, or correctness of the logic of the MODEL specification, a task which is performed by a later phase of the Processor. An example of a local semantics checking routine is one which checks the range of a numeric computation. For instance, if a group of data is said to occur n to m times, a subroutine exists to check the condition 0 <= n < m <32768. These manually-written routines are invoked automatically by the SAP by virtue of their specification in the EBNF/WSC of the MODEL language for the SAPG. The semantic checking routines are listed in Table 3.4.

<u>Semantics</u> <u>Cheching</u> <u>Routines</u>

NAME	WHAT IT DOES
ASSINIT	Initializes number of sources/targets to assertion
CATREC	Recognize the operator '||'
BITSTR	Check that an alleged bit string contains only the digits 0 and 1
CKMNMX	Checks proper range for mininum and maximum
EXPREC	Recognizes the operator '**'
FNCHECK	Check that a candidate name is a recognized function name
INITQNM	Initializes number components to qualified name
INITSFL	Initializes source file list
INITTFL	Initializes target file list
INTOASS	Returns 1 if the current scanned statement is an assertion and not a data description statement
INTODDL	Records that the statement scanned is a data description statement
INTREC	Recognizes integer
MEMINIT	Initializes number of members of record or group

Table 3.4 Semantics Checking Routines

Semantics Cheching Routines

NAME	WHAT IT DOES
MKQNM	Concatenates qualified name components
MOPREC	Recognizes a multiplication operation, i.e. '*' or '/'
NAMEREC	Name recognizer; checks not keywords
OPREC	Recognizer for the operators '+', '-'
OR_REC	Recognizes the alternation operation '|'
RECNUM	Recognizes and scans a number
RELREC	Recognizes any of the relations: <, >, >=, <=, ¬=, =
SETBIT	Used to set and reset a bit that indicate whether the statement is an assertion or a data description statement
STARREC	Recognizes a '*' for indefinite repetition
SVASSR	Saves the actual assertion itself during the scanning of a statement
SVENDC	Recognizes a ';' as an end of statement character

Table 3.4 Semantics Checking Routines

3.2.3 ERROR MESSAGE STACKING ROUTINE

There is a subroutine which stacks error diagnostics to print out upon recognition of a syntactically-incorrect user statement. Upon reaching incorrect syntactic units, the automatically generated SAP does not print its own messages, but expects the corresponding diagnostics to be on an "error stack". Specifically, an error code has to be stacked for each expected terminal symbol in the MODEL language in case the token is missing or incorrect. If the expected token is found, the SAP simply pops the corresponding error code and continues; if the expected token is missing or incorrect, the SAP pops the corresponding error code, prints the statement number, the unexpected token, and the corresponding error message, scans for the end of the statement delimiter (;), and continues. The routine that stacks such error codes is called "E". Each syntax error message pinpoints the token that is incorrect, missing, unexpected, or misspelled.

One product of the syntax analysis phase is the Error Diagnostics Report containing the error messages. Each mesage gives the diagnostics corresponding to the error code and provides the exact location of the error so that it can be corrected and resubmitted by the user easily. If no syntax errors are found during the syntax analysis phase, a message is sent that "NO ERROR OR WARNINGS DETECTED", and the Processor proceeds to the next phase. But if error diagnostics were produced, a flag is set to disable continuation of analysis and design beyond the syntax checking phase.

The error messages are listed in Table 3.5.

ERROR MESSAGES

CODE	ERRORS
1	A bit string contains character other than 0 or 1
2	Missing ':' after the word BLOCK
3	Badly formed boolean expresion after IF in statement
4	Missing or invalid numeric constant in iterative count spec
5	Missing or invalid numeric constant in relative error spec
7	Organization type missing or illegal in DISK statement
9	Type disk missing or illegal in DISK statement
12	Missing ELSE in comditional expression
14	Assertion missing after the keyword THEN
18	No boolean expression after the keyword IF
22	no expression after the keyword '('
23	Keyword '=' is missing
24	Missing right parenthsis
26	Missing string after quote
33	Error in recognition of a right hand side of an assertion
38	Keyword THEN is missing
39	Record or group keyword expected
42	Record name missing or illegal in FILE or REPORT statement

Table 3.5 ERROR MESSAGES

ERROR MESSAGES

CODE	ERRORS
44	Medium name missing or illegal in FILE or REPORT
45	Keyname missing in FILE or REPORT statement
46	Maximum length missing or illegal in variable length in FIELD statement
47	Invalid or missing field type in field/interim statement
48	Missing or invalid length in field/interim statement
49	Missing right parenthesis after field-type in field/interim
50	'-' sign is not succeded by an integer
51	Missing/invalid max no. of occurrences of items.
52	Name missing or illegal in item list
53	Missing left parenthesis in line spec
54	Missing integer in line spec
55	Missing right parenthesis in line spec
56	Missing/invalid file name after keyword FILE
57	FORMAT missing/misspelled after RECORD in storage statement
58	Missing/invalid tape label
59	Keyword RECORDSIZE missing/misspelled after MAX
60	Missing/invalid volume name (external or internal)
61	Missing/invalid device type
62	Missing/invalid iterative solution method

Table 3.5 ERROR MESSAGES

ERROR MESSAGES

CODE	ERRORS
63	Colon missing after keyword MODULE
64	Name missing or illegal in MODULE statement
65	Error in assembly of a number constant
66	Tape spec. parameter missing or illegal
67	Error in a picture spec
68	Qualified name illegal
69	Record format missing or illegal
70	Keyword BLOCKSIZE missing in record format spec
71	Blocksize value missing/illegal in record format spec
72	Record size value missing/illegal in record format spec
74	Missing ';' at end of statement
75	Missing ':' after keyword SOURCE FILES
76	Name missing/illegal in source file list
77	':' missing after keyword TARGET
78	Name missing/illegal in TARGET file list
79	Missing THEN in conditional expression
80	Unrecognizable statement
81	Badly formed arithmatic expression
82	Badly formed boolean expression
83	Badl formed boolean term
84	Badly formed concatenation of expressions

Table 3.5 ERROR MESSAGES

ERROR MESSAGES

CODE	ERRORS
85	Badly formed factor
86	Badly formed primary
87	Badly formed term
90	Missing left parenthesis in column spec
91	Missing integer in column spec
92	Missing right parenthesis in column spec
101	Length of picture spc. is too small or too big
102	Specified length is inappropriate for specified type of data
104	Specified maximum length is inappropriate or too small
105	The fraction point offset is outside of bounds -128<p<127
106	Bad repetition specification
107	Illegal character in picture specification
108	Expecting a level number in a structured data description statement

Table 3.5 ERROR MESSAGES

3.2.4 ENCODING USER STATEMENTS

These supporting routines encode some of the MODEL specification into an internal representation. Although all of the names provided by the user specification are kept intact in internal form for use by the object program, many of the descriptions and attributes are encoded for more compact and efficient processing later. For example, the description in a FIELD statement enters an internal table where the type of field is encoded (0 for character, 1 for binary, 2 for numeric, etc.), and the field length type is encoded (0 for fixed length, 1 for variable length). One encoding routine is written for each statement type. Each routine is invoked automatically after recognition of the syntactic unit by the SAP. The invocation is automatically generated as part of the SAP by the SAPG by virtue of its specification in the EBNF/WSC. The internal format of the tables is given in the next section in conjunction with the discussion of the internal associative storage of the MODEL statements.

The encoding and saving routines are listed in Table 3.6.

Table 3.6 ENCODING/SAVING ROUTINES

ENCODING/SAVING ROUTINES	
NAME	WHAT IT DOES
INITNUM	Initialize scanning a numeric constant
SETDEVB	Set device flag in media description to imply disk storage
SETDEVC	Set device flag in media description to imply that input is from cards
SETDEVP	Set device flag in media description to imply PRINTER
SETDEVT	Set device flag in media description to imply a terminal
SETDEVU	Set device flag in media description to imply a card punch
SETFUNC	Initiate a node in the syntax tree to store a function reference
SETNUM	Set for assembling a constant number
SETSTRN	Initiate a node in the syntax tree to store a string constant
SETSUBV	Initiate a node in the syntax tree to store a subscripted variable
SETVAR	Initiate a node in the syntax tree to store a variable name
STALL	Stores a node in the syntax tree after all its components have been defined

Table 3.6 ENCODING/SAVING ROUTINES

ENCODING/SAVING	ROUTINES
NAME	WHAT IT DOES
STBIT	Sets the current string contained in the temporary node to be a bit string
STDEV	Store device; Tape or Disk
STFUN	Stores a node in the syntax tree which contains a function name
STNUM	Concludes the assembly of a constant number
STPIC	Concludes the storing of a picture type specification
STR_CON	Stores a node in the syntax tree which contains a general constant
STRHS	Stores an assertion in the associative memory (an entry point in ASSINIT)
SVAAS1	Sets a node to contain a conditional assertion
SVASAE1	Sets to define a node containing a simple assertion
SVBEXP	Sets a node for storing a boolean expression
SVBF1	Sets a node for storing a boolean factor
SVBLK	Saves block size in disk/tape storage entry
SVBT1	Sets a node for storing a boolean term
SVCC	Encodes character code

Table 3.6 ENCODING/SAVING ROUTINES

ENCODING/SAVING NAME	ROUTINES WHAT IT DOES
SVCMP1	Save in a node the recently scanned syntactical unit as the first descendant
SVCOL	Saves column number in field storage entry
SVCON	Sets a node for storing a concatenation of expressions
SVCOND	Sets a node for storing a conditional exp.
SVDEN2	Saves density for tape
SVDEV	Set device name to storage name, and save device: Tape or Disk
SVFAC	Sets a node for storing a factor
SVFDTP2	Encodes field type, including NUM and DEC
SVFLD	Encodes field statement type as FLD
SVFLNM	Save file name. Call SVFILE, set default names for record storage, and reset device bit (DEVBIT)
SVKEY	Saves key field in file storage entry
SVLAB	Encodes label type in tape statement 0=none, 1=IBM_STD, 2=ANSI_STD, 3=BYPASS
SVLAB2	Save label for tape
SVLINE	Saves line number in field storage entry
SVMEN	Saves member name in record/group storage entry

Table 3.6 ENCODING/SAVING ROUTINES

<u>ENCODING/SAVING</u> NAME	<u>ROUTINES</u> WHAT IT DOES
SVMNFLN	Saves minimum field length in FLD statement
SVMNOC	Saves minimum number of occurrences in record or group storage entry
SVMOD	Marks the mode as FIXED or FLOATING
SVMXFLN	Saves maximum field length in FLD statement
SVMXOC	Saves maximum number of occurrences in record or group storage entry
SVNUMTP	Marks the data type as a numeric data type (BINARY or DECIMAL)
SVNXCMP	Saves the next assembled syntactical unit in a syntax node which is its ancestor
SVNXOP	Saves the next delimiter associated with the assembled syntactical unit or separating it from its successor
SVOP1	Saves an initial delimiter associated with phrase such as unary '-' or 'IF'
SVORG3	Saves organization for disk
SVPAR2	Saves parity for tape
SVPIC	Denote the data as 'PICTURE'
SVPRIM	Sets for assembling a phrase for a PRIMARY
SVPICST	Saves the picture specification string
SVQTY2	Saves quantity for disk

Table 3.6 ENCODING/SAVING ROUTINES

ENCODING/SAVING	ROUTINES
NAME	WHAT IT DOES
SVRCSZ	Saves record size in tape/disk storage enrty
SVRECF	Encodes record format on tape/disk storage; 0=FIXED, 1=FIXED BLOCK, 2=VARIABLE
SVSCALE	Saves the scale factor specified in the precision specification of the data type
SVSRC	Saves source file name in source storage entry
SVSTAR	Records and saves the repetition spec. '(*)' in a file statement
SVSTFL2	Save start file# for tape
SVSTNM	Saves storage name in FILE storage entry
SVSTRNG	Transfer an assembled string constant from the general buffer into a special temporary storage. The final storage of the node will be done by STR_CON.
SVTAR	Saves target file name in target storage entry
SVTERM	Initializes a node to store a phrase for a TERM
SVTRK2	Saves number of Tracks for tape
SVUCYL	Save units as CYL for disk
SVVOL	Saves volume name in disk/tape storage entry

3.2.5 STATEMENT STORAGE ROUTINES

These routines collect the strings of names and other vital information in the MODEL statements, and pass them to the STORE system, which is a subsystem in itself to store the statements for later processing. Such storage-invoking routines are called at the end of scanning each MODEL statement, and are the ones that begin with the letters "ST" (e.g. STFLD, STREC, etc). The storage subsystem described below (STORE), which is called by these routines, stores the MODEL statements in a simulated associative memory that facilitates later retrieval.

On analyzing the assertions (computational statements) a syntax or derivation tree which represents the assertion is generated and stored. This representation facilitates later analysis and scanning of the assertion, as well as systematic transformation. The tree representation is reconverted into text form in the code generation phase.

At the end of the syntax phase, we have the entire set of MODEL statements stored in a convenient storage system for further analysis. The storing subroutines which invoke the use of the STORE system act as an interface between the automatically generated SAP and the storage system presented below. The storage system is an extension to the capabilities of the SAPG since it is general purpose in nature and is independent of the nature of the language specified, and could be used for processing other languages.

The storing routines are listed in Table 3.7.

Table 3.7 STORING ROUTINES

STORING ROUTINES NAME	WHAT IT DOES
STFILE	Stores FILE statement
STFLD	Stores FIELD statement
STGRP	Stores GROUP statement
STMOD	Stores MODULE statement
STPNCH	Stores PUNCH statement
STREC	Stores RECORD statement
STSRC	Stores SOURCE FILES statement
STTAR	Stores TARGET FILES statement

3.2.6 HOUSEKEEPING ROUTINES

Finally, there are a few "housekeeping" type subroutines which need not be written by the language definer because they are provided by the SAPG, but which need to be included in the EBNF/WSC.

The housekeeping routines are listed in Table 3.8

Table 3.8 HOUSEKEEPING ROUTINES

<u>HOUSEKEEPING ROUTINES</u>

NAME	WHAT IT DOES
ADLEX	Adds a subpart of a floating point constant to its full representation
CLRERRF	Clears errors flag every statement to indicate no syntax errors yet in next statement
ENDINP	Executed upon end-of-file to print last line and wrap-up
FREETMP	Frees allocation of a temporary data structure which was needlessly allocated
NEGATE	Negates the value of a negative integer constant to derive its real representation
STMT_FL	Scans for end of statement delimiters when unrecognizable statement encountered
STMTINC	Increments the statement number; called at end of each statement

3.2.7 AN INDEX TO SAP ROUTINES

The subroutine names used in the specification of MODEL can be classified into one of the following four types of subroutines: encoding/saving routines, storing routines, semantics checking routines, and housekeeping routines. Table 3.6, 3.7, and 3.8 provide an alphabetical listing of the routines within each category. As for error messages, the error code and their meanings are shown in Table 3.5.

3.3 THE STRING STORAGE AND RETRIEVAL SUBSYSTEM

3.3.1 INTRODUCTION

The store routines that are referred to in the EBNF description of MODEL, utilize a general-purpose mechanism for storing source language strings. A similar mechanism is used later for retrieving these source language strings. The following system, basically, consists of a directory structure, described in section 3.3.2 and the format of storage entries described in Section 3.3.3. There are also two main procedures:

(1) STORE for storing source language string collected during syntax analysis. STORE is described in Section 3.3.4.

(2) RETRIEVE for accessing previously stored source language strings, based on a variety of "keys". RETRIEVE is described in Section 3.3.5.

Additionally a set of routines specified in EBNF parses and stores the assertions. Section 3.3.6 describes the format of stored assertions. Section 3.3.7 describes the routines that store the parsed assertions. These routines have also been referred to in the description of saving and encoding routines in Section 3.2.4.

The STORE procedure accepts strings which are formed by the subroutines called during syntas analysis. It stores the strings in memory which we call "storage entries" while building "directory entries" in a directory of certain names designated as keys. By building a directory, the strings are stored "associatively" in the sense that statements can later be retrieved based on their content. This capability is crucial to "non-procedural" language processor since the statements can be input in any order.

3.3.2 THE DIRECTORY AND STORAGE STRUCTURE

The storage entries (the strings to be stored) consist of two parts:
(1) the key names to be entered in the directory which include the names the user provided in the MODEL statements for naming data, assertions, etc. these are the names by which we may want to retrieve information later.
(2) auxiliary data from the source language strings including the encoded information in table form. This information is not used as the basis of retrievals.

Each storage entry will contain information from a given MODEL statement. They will appear in memory in the order in which they are processed.

The directory consists of an entry for each key name. Each directory entry points to the first storage entry containing that key name. A linked-list is then maintained from the first storage entry with that key name to other storage entries containing the same key name. A binary tree structure was chosen for the directory to make tree modifications and key names searches efficient. It is the first key name entered in the directory which becomes the root of the directory tree; the next key is entered "above" or "below" it in the tree by lexicographic order; etc.

Each directory entry has the following form:

Key name	Ptr-to-first	Up-pointer	Down-pointer

where "Keyname" is a string of (up to) 10 characters (padded with blanks to its right side)
"ptr-to-first" is a pointer to the first storage entry containing the "key name".
"up-pointer" and "Down-pointer" are pointers to other directory entries, whose key names are up or down, respectively, in the lexicographic sense.

Each storage entry has the following form:

```
| N | name-1 | ptr-1 | . . . | Name-n | Ptr-n | ptr-to-data|
```

where N is the number of key names in the storage entry string.
Name (i=1 to n) is a key name of a variable.
Ptr (i=1 to n) is a pointer to the next storage entry with the same key name.
Ptr-to-data is a pointer to auxiliary data from the source language statement.

Figure 3.4 depicts an example of three storage entries and a directory consisting of only three entries, X, Y, and Z, where Y is the root of the directory tree. Such a structure was partially motivated by similar ideas in the "Multi-list" file organization.

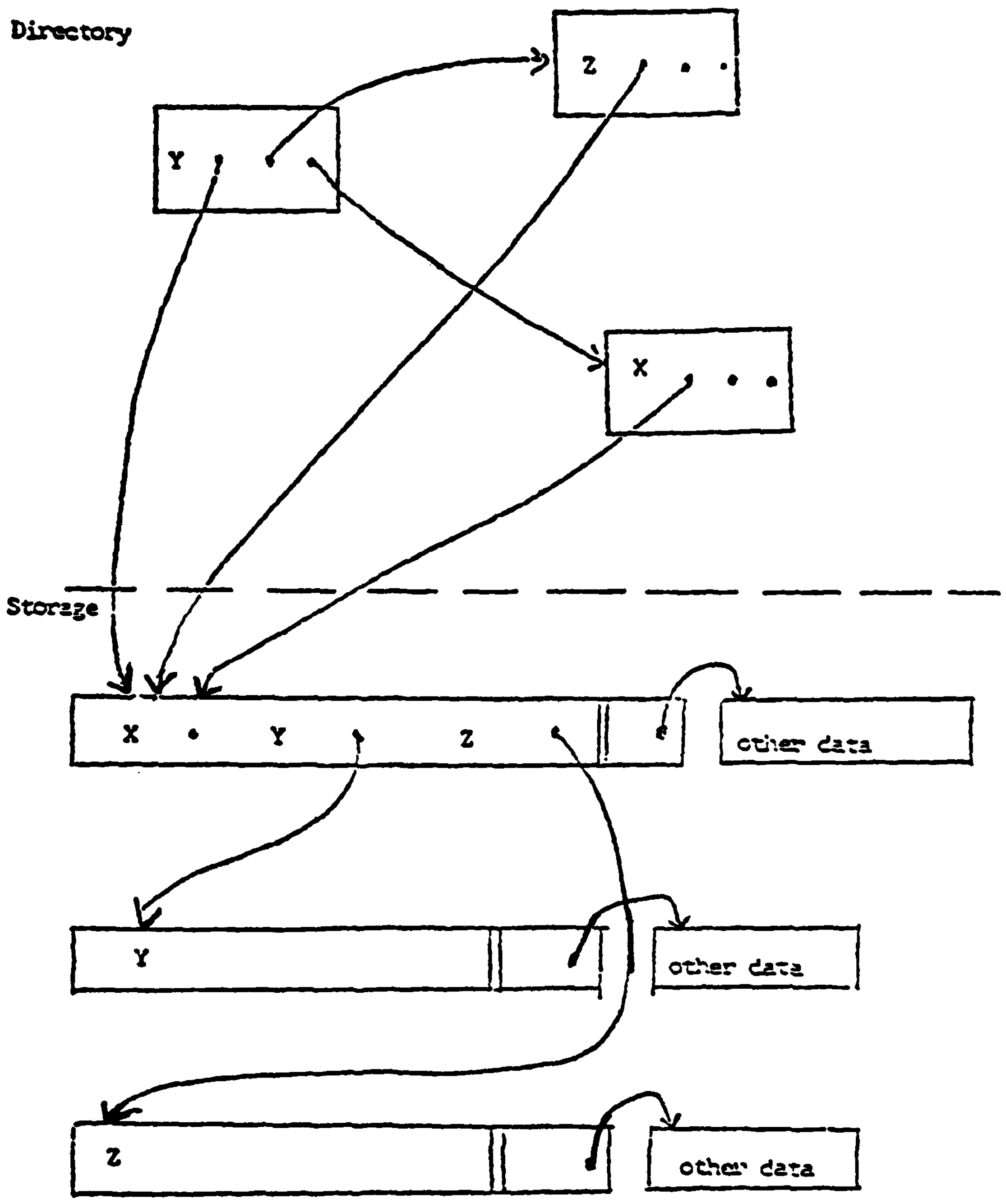


Fig. 3.4 Sample Directory and Storage Enties

3.3.3 STORAGE ENTRIES FORMAT FOR MODEL STATEMENTS

The STORE mechanism, described in the next section, is called by SAP's storing subroutines to store the MODEL statements for retrieval (by RETRIEVE) in the later phases. For each type of MODEL statement, the key names in it are stored in its storage entry. The non-key information in the MODEL statement (information which is not used to specify retrievals) is kept in description tables, which are connected (by STORE) to the corresponding storage entries as was shown above. Table 3.9 summarizes the internal format of the storage entries and the corresponding description tables for each type of MODEL statement. The leftmost name in each entry is the name of the statement being stored. The middle column shows the information appearing in the corresponding storage entry (with the pointers omitted due to lack of space). The right column shows the additional encoded information, if any, from the statement. The key names beginning with a dollar sign ($) in the storage entries are not user-proveded, but are inserted by the system for its own information. The last name in each storage entry, for example, identifies the type of statement.

Table 3.9 Storage entries Format for MODEL

MODEL Statement Schema	Storage Entry Key Names	Auxilliary Descriptions				
		Type	Stmt#.			
MODULE: module-name	module-name \$MODULE	MODL	n			
SOURCE FILES: s_1, s_2,...,s_n	\$SOURCE s_1 s_2 ... s_n	SRCP	n			
TARGET FILES: t_1, t_2, ... t_m	\$TARGET t_1 t_2 ... t_m	TARP	n			
filename IS FILE((GROUP / RECORD) is r(*) STORAGE IS s, KEY IS k, ORG IS 0)	filename r s k \$FILE:	FILE	n	ORG-Code 0- SAM 1- ISAM	Key-flag 0 no sort key 1-sort key	is-star 0-no repet. for r 1-r repeat:
record-name IS RECORD (m_1, m_2,...,m_n)	record-name m_1 m_2 ... m_n \$Pfile \$RECD	RECD	n	#members	members #subscripts first sub. second sub.	
group-name IS GROUP (m_1,m_2,...,m_n)	group-name m_1 m_2 ... m_n \$Pfile \$GRP	GRP	n	(same as record)		

Table 3.9 Storage entries Format for MODEL

MODEL Statement Schema	Storage Entry Key Names	Auxiliary Descriptions				
		Type	Stmt#.			
field IS FIELD (fieldtype (minlength : maxlength)	fieldname $Pfile $FLD	FLD	n	fieldtype 0-char 1-binary 2-numeric 3-decimal 4-binary floating 5-bit 6-decimal floating 7-picture	length type 0-fixed 1-variable	min/max scale factor picture string (if type =7)
SOURCE: $a_1, a_2, \ldots, a_n$	assertion-name a_1 $a_2 \ldots a_n$ $ASSERT	ASSR	n	#names components		
TARGET: $t_1, t_2, \ldots, t_m$	assertion-name t_1 $t_2 \ldots t_m$ $ASSERT	ASTG	n	#names components		
assertion-name:	assertion-name $ASSERT	ASTX	n	Pointer to syntax tree		
subscript-name IS SUB[SCRIPT][(range)]	subscript name $$SUB	$SUB	n	range		
storage-name IS						
CARD	storage-name $CARD	CARD	n			
TAPE (...)	$TAPE	TAPE	n	tape-attributes		
DISK (...)	$DISK	DISK	n	disk-attributes		
TERM (...)	$TERM	TERM	n	term-attributes		
PUNCH(...)	$PNCH	PNCH	n	punch-attributes		
PRINTER (...)	$PRNT	PRNT	n	print-attributes		

3.3.4 THE STORE PROCEDURE

The STORE(S,D) Procedure has two parameters, S and D. S is the string containing the key names which are to be stored and to be entered in the directory. D is a pointer to previously built auxiliary data from the source string. The latter usually is an encoded form of non-key source language information.

Algorithm STORE shows the storing procedure. STORE receives the key names from S and creates a storage entry for it (Steps 1-3). It checks if they are in the directory (Steps 4-5, subroutine SEARCH DIR). If the key is in the directory, then it follows the "pointer-to- first" :h points to the first storage entry with that name (Steps 7-8). The array of strings in that storage entry is scanned until the key name is found. If its "nest" pointer is null (end-of-list), then it is set to point to the newly created storage entry (Steps 8-11). If it is not, the process is repeated until a null (end-of-list) pointer is found (Steps 9-10). If the current key name is not found in the directory, it is entered in the appropriate spot in the lexicographical position in the directory (Step 6, subroutine CREATE DIR) and the pointer in the directory is set to point to the newly created first storage entry (Steps 7-8).

Algorithm STORE : The Store Procedure

Parameters: S=string of keys to be stored;
D=pointer to other data

(see Section 2.3.2 for diagrams of Data Structures)

[Subroutines called: CHECK_DIR, GENERATE_ENTRY]

Step 1. Count #KEYS.

Step 2. Allocate the storage entry for S (call it SE, according to the format shown).

Step 3. Connect PTR_TO_DATA in SE to D.

Step 4. For each key name, perform steps 5 through 11.

Step 5. If key exists in the directory (Algorithm CHECK-DIR), then go to step 7; else go to step 6.

Step 6. Create a directory entry for this key. (Algorithm GENERATE-ENTRY)

Step 7. Let DE=this directory entry.

Step 8. If PTR_TO_FIRST in DE already points to a first storage entry with this key name, then go to step 9; else go to step 11.

Step 9. Get the next storage entry in the list.

Step 10. If it is the last in list, then go to step 11; else go to step 9.

Step 11. Add the new SE to the list.

Step 12. Return.

3.3.5 THE RETRIEVE PROCEDURE

RETRIEVE(E,D,S,N,P) is the procedure for retrieving desired storage entries, by searching through the data structures depicted in Figure 3.4 and Table 3.4. It is invoked by many routines described in subsequent phases of the Processor. It has five input parameters as indicated. RETRIEVE finds all the storage entries in which the given key name or expression of key names, E, appears and furthermore checks whether the first characters of data associated with the storage entries match the string D. That is, RETRIEVE finds all the storage entries with keys satisfying the logical expression E and other data D. RETRIEVE starts its search at directory entry S, normally the root node of the directory, and it returns a list of pointers P, to those storage entries which satisfy the request of the calling program. The number of storage entries satisfying the request is returned in N.

The logical expression used to retrieve strings can be any boolean expression involving "key" names or names in the MODEL statements in disjunctive normal form, where the first key in each term is non-negated. For example, consider the following statement by a calling program:

```
        CALL RETRIEVE(KEYS, '', START, N, P);
```

KEYS might contain the string value 'PRICE & ^QUANTITY |EXTENT '. This makes RETRIEVE find all storage entries (which correspond to all statements in the MODEL specification) in which PRICE appears and QUANTITY does not appear, or statements in which EXTENT appears. The null second parameter means that the auxiliary data portion of each statement is immaterial. RETRIEVE would then start its search and return a list of pointers in P to those storage entries which satisfy the condition, and N would be set to the number of statements that satisfy the condition.

Algorithm RETRIEVE is shown in the following page.

Algorithm RETRIEVE : The Retrieve Procedure

Parameters: E=logical expression string; S=pointer to beginning of directory (input);
P=list of pointers satisfying E; N=number of satisfying entries

(see Figure 7a for diagrams of data structures)

Step 1. Get leading key name K of next conjunct from E. If no more, go to Step 22.
Step 2. Check directory for K (standard binary tree search in subroutine SEARCH-DIR given earlier).
Step 3. If found, then go to step 4; else go to step 1.
Step 4. Set PSE=PTR_TO_FIRST (pointer to first storage entry with K)
Step 5. Add PSE to W list (temporary list of pointers)
Step 6. If K in PSE storage entry points to another storage entry with K, then go to step 7; else go to step 8.
Step 7. Set PSE to next storage entry in the list, go to Step 5.
Step 8. If end of E, then go to step 20; else go to step 9.
Step 9. Get next symbol in E.
Step 10. If symbol='&' then go to step 14; else go to step 11.
Step 11. If symbol='|' then go to step 12; else error return.
Step 12. Add list of pointers in W to list of pointers in P without duplication.
Step 13. Go to step 1.
Step 14. Get next symbol.
Step 15. If symbol='~' then go to step 16; else go to step 18.
Step 16. (Case of conjoining negated term) eliminate pointers in W to storage entries which also contain next key name in E.
Step 17. Go to step 8.
Step 18. (Case of conjoining non-negated term) eliminate pointers in W to storage entries which do not contain next key name in E.
Step 19. Go to step 8.
Step 20. Add list of pointers in W to list of pointers in P.
Step 21. Set N=number of pointers in P list.
Step 22. Return.

An example showing the retrieval mechanism to retrieve all storage entries with key names "B" and "C" is given in Figure 3.5.

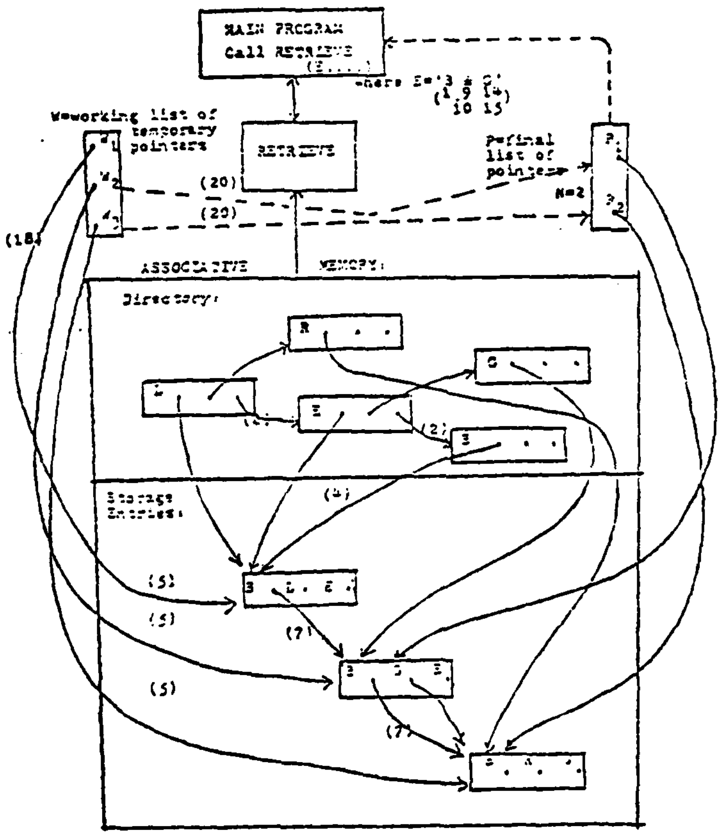


Fig. 3.5 Example of Retrieval Mechanism

The diagram shows in parentheses the steps that correspond in the algorithm. RETRIEVE starts by getting the leading key name of the first conjunct (Step 1) and searches the directory for it (Step 2). If found, it puts the list of pointers to all storage entries with that name in a temporary list (Step 3-7). If there are other names in the conjunct (Steps 10, 14), then RETRIEVE eliminates from the temporary list those pointers whose storage entries do not have the other names in the conjunct (Steps 14-16). If there are more conjuncts in the expression, then the process is repeated and additional pointers are added to the list (Steps 12-13). When the end of the expression is reached, the list of pointers to the satisfying storage entries and the number of pointers are returned (Steps 20-22).

3.3.6 STORAGE STRUCTURES FOR ASSERTION STATEMENTS

Analysis of an assertion statement causes two storage entries to be made for the satatement. (See also Table 3.9). The first entry has the type ASTX and contains in its main part just the assertion label (system generated) and a keyword $ASSERT. Its auxiliary data contains a pointer to the syntax tree which represents in a parsed form the body of the assertion. The second entry has the type ASTG and contains a list of all the names which are sources and targets to the assertion. Sources are all the names which appear on the right hand side of each equal sign, (including subscript expressions) and within boolean condition expressions. Targets are the names whose values are defined by the assertion.

3.3.6.1 THE SYNTAX TREE FOR AN ASSERTION

The syntax tree of an assertion is constructed out of mutually linked nodes. There are nodes of two types: non-terminal nodes which have descendants and terminal nodes which have no descendants and represent atomic syntactical units such as identifiers, numeric and string constants. Each node corresponds to a phrase in the parsed assertion, and if it is non-terminal the list of its descendants represents the further breakup of this phrase.

3.3.6.2 THE STRUCTURE OF NON-TERMINAL NODES

The structure of non-terminal nodes is as follows:

TYPE	n= Number of Sons	Delimit #1	Pointer to Son1 #1	...	Delimit #n	Pointer to Son #n

where "TYPE" is an integer code identifying the syntactical type of the phrase according to the following legend:

o - Conditional Assertion. Example: if A=B THEN C=D
1 - Simple Assertion. Example: A=B
2 - Conditional Expression.
    Example: IF A > B THEN C ELSE 0
5 - Boolean Expressions. Example: (A=B) | (C=D)
6 - Boolean Term. Example: (A > 5) & (C <= 3)
7 - Boolean Factor. Example: C = 7
8 - Concatenation. Example: All || 'END'
9 - Arithmetical Expression. Example: (A*B)+(C*D)

10 - Term. Example: A*B
11 - Factor. Example: A**2
12 - Primary. Example: A, B(I+1), (A+B)
13 - Function. Example: SUM(A,I)
14 - Subscripted Variable. Example: A(FOR_EACH.A)

"Number of Sons" is the number of components or subphrases that the indicated phrase is broken into. Thus if the phrase is "A+B" it is of type 9 (Arithmetical Expression) and it is parsed further into the subphrases "A" and "B". The '+' delimiter will be stored as delimiter number 2 in the current node.

The delimiters are encoded as integers according to the following legend:

1 - ' '(Blank - No delimiter)
2 - 'IF' (keyword)
3 - 'THEN'
4 - 'ELSE'
5 - '='
6 - '+'
7 - '-'
8 - '*' (Standing for multiplication)
9 - '/'
10 - '**' (Exponentiation)
11 - '|' (Alternation - Logical 'or')
12 - '_
13 - '||' (Concatenation)
14 - '' (Negation)
15 - '('
16 - ')'
17 - ','
18 - '>'
19 - ' >='
20 - '<'
21 - '<='
22 - '='
23 - '>'
24 - '<'

"Delimiter 1, i=1, ..n" are the delimiters separating the subphrases. The first one is the delimiter prefixing the whole phrase such as the '-' in the phrase -A or the ' ' in the phrase ' (A<B & B<C)'. "Pointer to Son i, i=1,..n" are pointers to other nodes which represent the subphrases into which the current phrase is parsed.

3.3.6.3 THE STRUCTURE OF TERMINAL NODES

Terminal nodes are used to store constants such as variable names, string or numeric constants. Their structure is as follows:

type	str-length	value

where "type" is an integer code identifying the type of the constant according to the following legend:

20 - character string constant. Example: 'ABC'
21 - function name. Example: SUM
22 - numeric constant. Example: 3.14
23 - variable name. Example: PAY
24 - bit string constant. Example: '1001'B

"Str-length" is the length of the character string representing the constant. It will be 3 for storing the variable name 'PAY'. "Value" is the actual character string representing the constant.

During later processing (Module ENEXDP), all the terminal nodes which refer to non-constants (types 21,23) are converted to a different format; referred to as variable-terminal-nodes:

Type	Node#

'Type' as before is an integer code identifying the type of the name according to the following legend:

25 - Variable type. The associated name is a variable and NODE_ is the dictionary entry number of this variable.
26 - Subscript type. This stores the name of a subscript. NODE# refers to a dictionary entry number. This dictionary entry can be of one of the following types:
'GRP', 'RECD', or 'FLD', which must be repeating. If this entry name is X then the name of the subscript is FOR_EACH.X.
'$SUB" - This is a global subscript declared by the user.
'$' - This is a free subscript added by the system. It is one of the subscripts $1..to$9.
27 - Function Name. NODE# is an index in a list of functions recognized by the system. See Table 3.10 for the list.

An overall example consider the syntax tree for the assertion:

```
  If A=B | C<D & E<=F
     THEN  X(FOR_EACH.X) = (Y+Z)*T|| '$';
     ELSE  X(FOR_EACH.X) = '0';
```

It is described in Fig. 3.6, with the modification that delimiters are represented by themselves rather then in their encoded form, to improve readability.

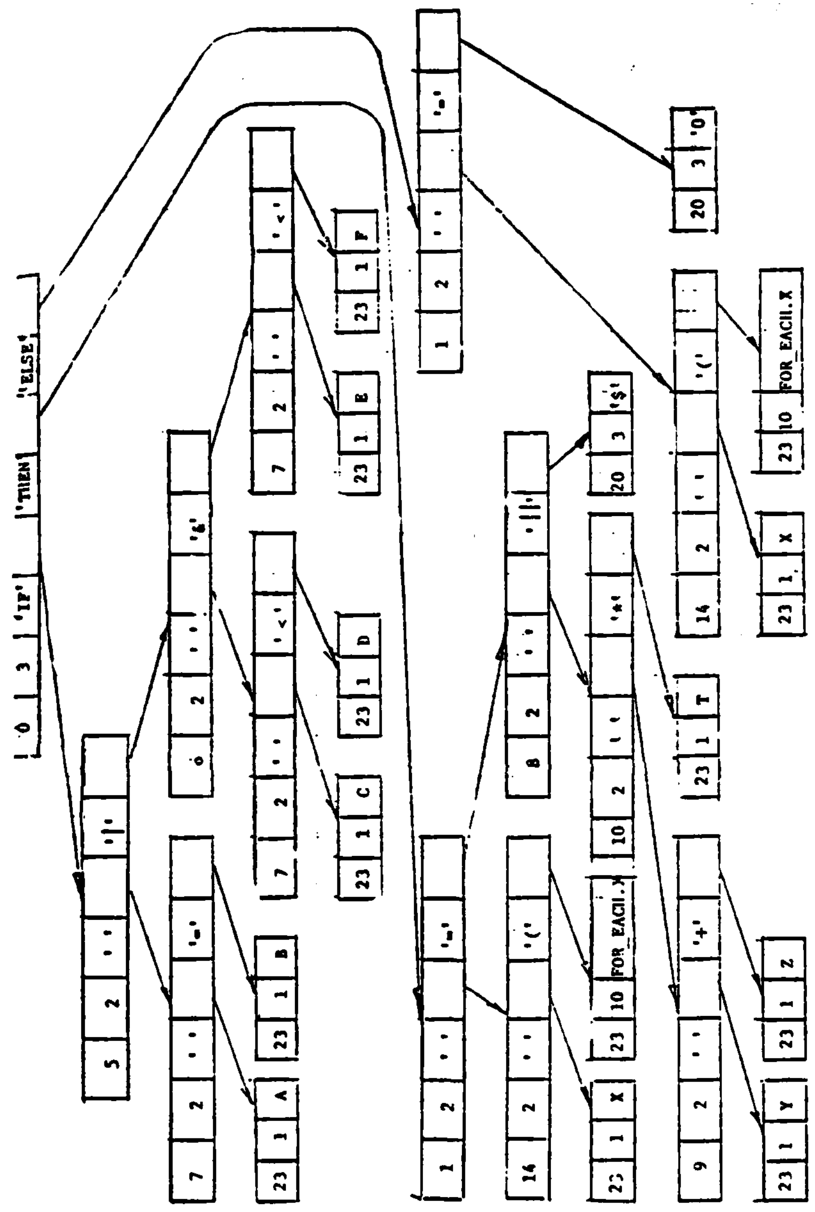


Fig. 3.6 Syntax Tree For Example - Assertion

3.3.7 THE SYNTAX TREE CONSTRUCTION ROUTINES

Several routines are responsible for the construction of the syntax tree of an assertion. They may be classified and described as follows:
Setup Routines: On entering a parse for a phrase of a certain type (by SAP) an appropriate setup routine is called. This routine allocates a temporary node area (temporary since we do not know yet how many subphrases or components it will have), assigns a type number corresponding to the type of the phrase and resets a component count to 0.
There is a setup routine corresponding to each phrase's type. They are for the non-terminal types (listed in increasing type code order):

SVAASO. SVASSR (SVASAE1), SVBEXP, SVBT1, SAVF1,
SVCON, SVAE, SVTERM, SVFAC, SVPRIM, SETFUNC, SETSUBV.

For the terminal types (codes > 19), a string area is allocated and a type variable is assigned, too. No setup routine exists for bit string since the distinction between it and a character string can be made only at the end of its scanning.
Save Routines: These are common to all non-terminal phrases. They alternately store delimiters and pointers to components, increasing the "number of sons" counter appropriately. These are all stored in the temporary node storage area.

SVOP1 - Stores a first delimiter. If this routine is not called the first delimiter is always set to 1 (= ' ').
SVCMP1 - Stores a pointer to the first component.
SCNXOP - Stores the recently scanned delimiter in the next available delimiter slot. Then increment the "number of sons" counter.
SVNXCMP - Stores a pointer to the recently assembled subphrase in the next available component slot.

Storing Routines: These finalize the node structure, after scanning of the phrase is complete. Since size of strings and number of sons are known by this time, a permanent node space is allocated and the contents of the temporary storage entry transferred there. The temporary storage area is then freed.

STALL - This is the storing routine for all the non-terminal nodes. It first checks to see if the assembled node is not trivial. It will be trivial if it contains only one component and the first delimiter is blank. In this case no permanent storage is made for this node. This check eliminates redundant nodes in the syntax tree. If the node is not trivial, a permanent allocation is made for it and the proper contents transferred there.
For the terminal nodes we have separate storing routines:

STNUM - Stores a numeric constant
STFUN - Stores a function name
SVSTRNG - Transfers a string constant to the storage area before calling on STR CON.
STBIT - Stores a bit string
STR_CON - A common routine for storing all constants. It allocates a permanent node storage and transfers type, length and string into it.

ABS	ADJR	AMAX	AMIN
ANY	BIT	CEIL	CHAR
COPY	DATE	DECIMAL	EXP
FALSE	FIXED	FLOAT	FLOOR
HIGH	INDEX	LENGTH	LOG
MAX	MIN	MOD	PAGE
REPEAT	ROUND	RTRIM	RUNSUM
SELECT	SIGN	SSN_FN	STRING
SUBSTR	SUM	TIME	TRANSLATE
TRUE	UNSPEC	UPDATE	VERIFY

Table 3.10 The functions recognized by the MODEL processor.

CHAPTER 4

PRECEDENCE ANALYSIS

4.1 INTRODUCTION

A MODEL specification consists of many data description or assertion statements. In principle, the data description statements specify the structure of data entities such as file, group, record, and field. The assertions specify the relationships between the data entities. The data entities and the assertions are referred to here as program entities. On the other hand, in an executable program there are program events such as I/O activities, computations, or getting data ready. The events in a program generated by the MODEL system correspond to entities in the specification. For example, a file entity corresponds to an event of opening a file or closing a file; a record entity corresponds to reading a record or writing a record; and an assertion entity corresponds to computing a target variable. The sequence of the program events is not given by the user. Instead, it is determined by the MODEL processor under the constraints of precedence relationships among the program events. In this chapter we discuss the analysis for recognizing the precedence relationships between program events and representing them in a directed graph.

Based on the specification we can find the unique symbolic names assigned by the user to data entities. Additionally the MODEL processor automatically assigns a unique name to every assertion. Similar to other compilers, the MODEL processor maintains a symbol table called dictionary which contains all the symbolic names of program entities and their attributes.

The dictionary is created by a procedure CRDICT which finds all the entities in the program specification and stores their names into the dictionary. Except for some special cases described below, there is a correspondence between each statement in the specification and an entity in the dictionary.

Attributes of a symbol such as the type (file, group, field, ..., etc), the number of dimensions, the structural relation of it to other symbols are stored in the dictionary during the process of precedence analysis, and later during dimension analysis. This information is used later to determine the execution sequence.

Various types of relationships among program entities have direct implication on the execution sequence of their corresponding program events. The precedence relationships among the program events are found based on the analysis of the program entities. For example, a hierarchical relationship exists when one data entity contains another, such as when a file contains a record, a record contains a field, ..., etc. A dependency relationship exists between a field and an assertion when the field is either a source variable of the assertion or its target variable. There are also relationships between data entities and their associated control variables. The events and their precedence relations are represented by a directed graph called an Array Graph.

The Array Graph is created by two procedures, ENHRREL and ENEXDP. The ENHRREL routine analyzes data description statements and finds the precedence relations caused by the hierarchical relations between data entities. The ENEXDP routine analyzes assertions and finds the precedence relations from the dependency relations among data fields and assertions. It also finds the precedence relations among data entities and their associated control variables. Since the Array Graph contains the complete precedence information, it is used to check the completeness and consistency of the specification and to determine the computation sequence.

4.2 REPRESENTATION OF PRECEDENCE RELATIONSHIPS

4.2.1 DICTIONARY

Every program entity has a full name which uniquely identifies it. Most of the entities have a single component full name. When two data entities share the same name, it is necessary to qualify the name with their respective file names to distinguish them. Two data entities within one file are not allowed to share the same name. A file name may have at most two instances denoted as NEW or OLD followed by an identifier. Thus a data entity may have a full name of three components: NEW or OLD, file name, and data name. Control variables have one component more than the associated data entities, i.e., a reserved key name. The full name and the attributes of each program entity are stored in the dictionary.

In order to use memory efficiently, memory space for the entries of the dictionary are allocated dynamically. Pointers to the dictionary entries are stored in a vector DICTPTR and the total number of pointers in the vector is denoted as DICTIND. With this arrangement, we can allocate memory piecewise and access the information randomly. Since each program entity corresponds to a node in the Array Graph, we will call its entry number in the dictionary node number. The organization of the dictionary is shown in Fig. 4.1 and the attributes in the dictionary are listed in Table 4.1.

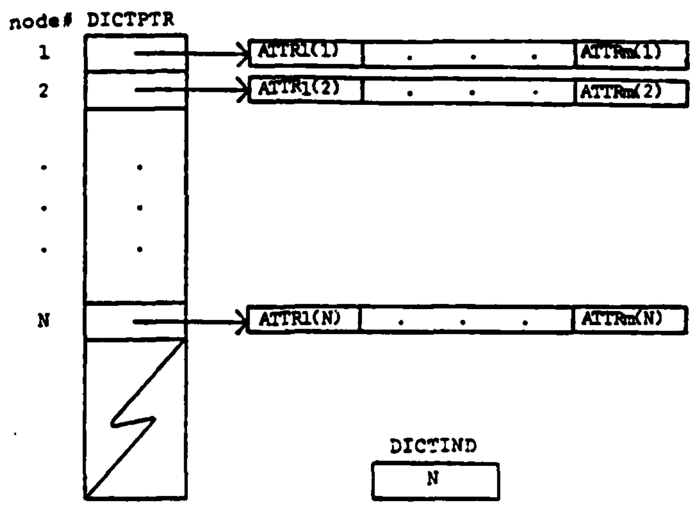


Fig. 4.1 Organization of the dictionary

Table 4.1 Attributes in the Dictionary

XDICT - Is the full name of the entity.
XNAMESIZE - Is the number of characters in XDICT field.
XUNIQUE - Is the smallest name by which the entity can be identified uniquely. If the file name component of a full name is not necessary to identify the entity uniquely, then XUNIQUE is set to the name without file name component; otherwise, XUNIQUE is set to XDICT.
XDICTYPE - Specifies the type of the entity. Following are the possible values:

- ASTX - An assertion.
- GRP - A group.
- FILE - A file.
- RECD - A record.
- MODL - The specification name.
- SPCN - A special name prefixed with a keyword such as END, SIZE, LEN, POINTER, NEXT, SUBSET, ENDFILE, and FOUND.
- $SUB - User or system declared subscripts, including the standard subscripts: SUB1, SUB2, ..., SUB10.
- $$ - System added subscripts: $1, $2, ..., $10.
- $$I - System loop variables: $I1, $I2, ..., $I10.

XMAINASS - Contains a pointer to the storage of the statement which defines the entity.

Table 4.1 Attributes in the Dictionary (Continued)

XNRECS	This count is meaningful only for file entities and holds the number of different record types contained in the file.
XPARFILE	Holds the node number of the parent file entity for all input and output data items.
XPAREC	For data items below the record level this field holds the node number of their parent record entity.
XINP	Is '1'B if the entity is in input file, and '0'B otherwise.
XOUP	Is '1'B if the entity is in output file, and '0'B otherwise.
XISAM	Is '1'B if the entity is an ISAM file, and '0'B otherwise.
XKEYED	Is '1'B if the data entity is in a file for which a key name was specified.
XLEN_DAT	The length in bytes of the data entity.
XREPTNG	Is '1'B if the data entity is repeating.
XVARYREP	Is '1'B if the data entity has a varying number of repetitions.
XMAX_REP	The maximal number of repetitions which was declared for the data entity. If no maximal repetition is declared, XMAX_REP is set to 1.
XVARS	Is '1'B if the entity contains a descendant below the record level and the descendant has a variable structure.

Table 4.1 Attributes in the Dictionary (Continued)

XSUBREC - Is '1'B if the data entity is a member of some record type.
XISSTARRED - Is '1'B if the data entity is repeating and has a undetermined repetition.
XFATHER - The node number of the data entity which is one level above the current entity in the data structure.
XSON1 - The node number of the leftmost descendant of the current entity.
XBROTHER - The node number of the immediate right neighbor of the current entity in the data structure.
XENDB - The node number of the control variable END.X if the currnt entity is X.
XEXISTB - The node number of the control variable SIZE.X if the current entity is X.
XVIR_DIM - The conceptual (virtual) dimensionality of the entity.
XSUBSLST - A pointer to the node subscript list associated with the entity.
X$SUCCESSORS - The number of edges in the XSUCC_LIST.
XSUCC_LIST - A pointer to the list of edges emanating from the current entity.
X$PREDECESSORS - The number of edges in the XPRED_LIST.
XPRED_LIST - A pointer to the list of edges coming into the current entity.

4.2.2 THE ARRAY GRAPH

The Array Graph is a directed graph which represents the precedence relationships among program events. The nodes in the Array Graph are the program events and the edges are the precedence relationships. One program event in the Array Graph will correspond to one program entity. Thus the nodes in the Array Graph correspond to the program entities in the dictionary. The edges between nodes are stored in edge lists associated with those nodes. The attribute SUCC_LIST of a node contains a list of edges emanating from it and the attribute PRED_LIST contains a list of edges terminating at this node. We can thus find the successors as well as the predecessors of any node.

The nodes in the Array Graph are compound nodes, i.e., an entire array of data is represented by one node. Also each assertion is represented by one node, independently of how many array elements it defines. The range of each dimension of a compound node is stored in the node subscript list associated with the node. The edges in the Array Graph are compound edges which denote arrays of relations between two compound nodes. With each edge are also stored the types of subscript expressions used in the relations between the source and the target node of the edge. The meaning of the Array Graph is made more precise by considering the corresponding Underlying Graph (UG), where every array element is represented by one node. An assertion node in the Array Graph may be expanded in the UG into as many nodes as the elements of the array which it defines. Edges are drawn between the simple nodes. The UG may be an enormous graph which is impractical to analyze. Sometimes the actual number of array elements is not known until run time. Thus it is impossible to create the UG of the specification. In contrast, the Array Graph is more compact and easy to analyze.

4.2.2.1 DATA STRUCTURE OF EDGES

Every edge from a node S to a node T has a uniform format:

$$T(U1, \ldots, Uk) \xleftarrow{t} S(J1, \ldots, Jm)$$

where t is the type of the edge,
k is the dimensionality of node T,
m is the dimensionality of node S,
Ji, 1<=i<=m, are subscript expressions appeared on the ith dimension of node S.
Ui, 1<=i<=k, are the node subscripts associated with the node T.

The subscripts U1, ...,Uk of the target node T are stored in the attribute XSUBSLST of T in the dictionary. Therefore they are not specified in the edge. In the later discussion, a type 4 subscript expression Ji will be indicated by an '*' in the ith dimension of the source node.
An edge is represented by the following data structure:
SOURCE : The source node of the edge.
TARGET : The target node of the edge.
EDGE_TYPE : The type of the edge.
DIMDIF : The difference between the dimensionality of the target node and the source node.
SUBX : A pointer to the subscript expression list (J1,...,Jm).

4.2.2.2 DATA STRUCTURE OF SUBSCRIPT EXPRESSION LIST

A subscript expression Ji can be classified into one of the following seven categories according to its composition (refer to section 3.3.2). Type 4 subscript expression is referenced later as a general subscript expression. Types 5, 6, and 7 subscript expressions are added for the efficient implementation of some list type functions[PNPR 80]. They are basically of the form X(I) where X is a variable but used to subscript another variable B in B(X(I)). This form of subscript expression is referred to as indirect indexing. The array used in indirect indexing must be integer valued with non-negative entries. The system will analyze indirect subscripts only if the indirect indexing array X(I) is sublinear, namely if it is:
a) Monotonic, i.e., if I>J then X(I) >= X(J).
b) Grows more slowly than I, i.e., X(I) <= I.

The system can test the indirect indexing array automatically to determine if it is sublinear by the following simple criteria. In the assertion that define the indirect indexing array X(I), the value of the right hand side must be either 0 or 1 for I=1 and must be equal to X(I-1) or X(I-1)+1 for I>1. Thus the system will examine the assertion to check if it is in the form:

```
  X(I) = IF I=1 THEN (1 | 0)
                ELSE (X(I-1) | X(I-1)+1) ;
```

An element in a subscript expression list is defined by the following data structure:

NXT_SUBL : A pointer to the next element of the list.

LOCAL_SUB$: If the subscript expression is of the form Uq[-c] or X(Uq[-c])[-k], then LOCAL_SUB$ is q, i.e. the ordinal number of the subscript Uq as it appears in T(Uk,...,U1).

APR_MODE : The type of subscript expression.

INXVEC : The node number of the indirect indexing vector X if the APR_MODE is 5, 6, or 7. Otherwise, 0.

4.3 CREATION OF THE DICTIONARY (CRDICT)

The procedure CRDICT analyzes the statements of the specification and enters all the program entities into the dictionary. To find all the data entities we start from the top level of data structures and then trace down the structures. The structures whose root is a file listed in the SOURCE FILE or TARGET FILE statements of the program header are considered external files, i.e. input file or output file. If a data structure is not part of any input or output file, it is considered an interim variable which is computed as any variable in an output file but not written to the external storage.

Corresponding to each input or output file, there is a file entity entered into the dictionary. If a file named F is served both as a source and a target file, then two file entities named OLD.F and NEW.F will be entered into the dictionary. Starting from the file entity we can find its immediate descendants from the file description statement, and the descendants' names will be prefixed by the file entity's name. If the root of a data structure is not a file, we will consider INTERIM as its file name and all the decendants will be put into dictionary, too.

As we analyze a data structure, we also construct a tree representation for it. For every data node we store pointers to its father, leftmost son, and younger (i.e. immediate to its right side) brother in the attributes XFATHER, XSON1, and XBROTHER respectively. We will illustrate this with an example in Fig. 4.2.

```
X IS GROUP (Y,Z) ;

Y IS FIELD ;

Z IS FIELD ;
```

$$\begin{tikzcd} & X \arrow[dl] \arrow[dr] & \\ Y & & Z \end{tikzcd}$$

```
X = XFATHER(Y)

X = XFATHER(Z)

Y = XSON1(X)

Z = XBROTHER(Y)
```

Fig. 4.2 Tree representation of data structure

After all the data entities are entered into the dictionary, a simplified name is derived for every data entry. If the file name component can be omitted from the full name without causing any ambiguity, the simplified name is the reduced name. Otherwise the simplified name is the same as the full name.

Other types of program entities such as module name, assertions, and subscript variables are defined by a specific type of statement respectively and there is a one-to-one correspondence between the statements and the entities. We can retrieve these types of statements from the associative memory and enter the entities into the dictionary.

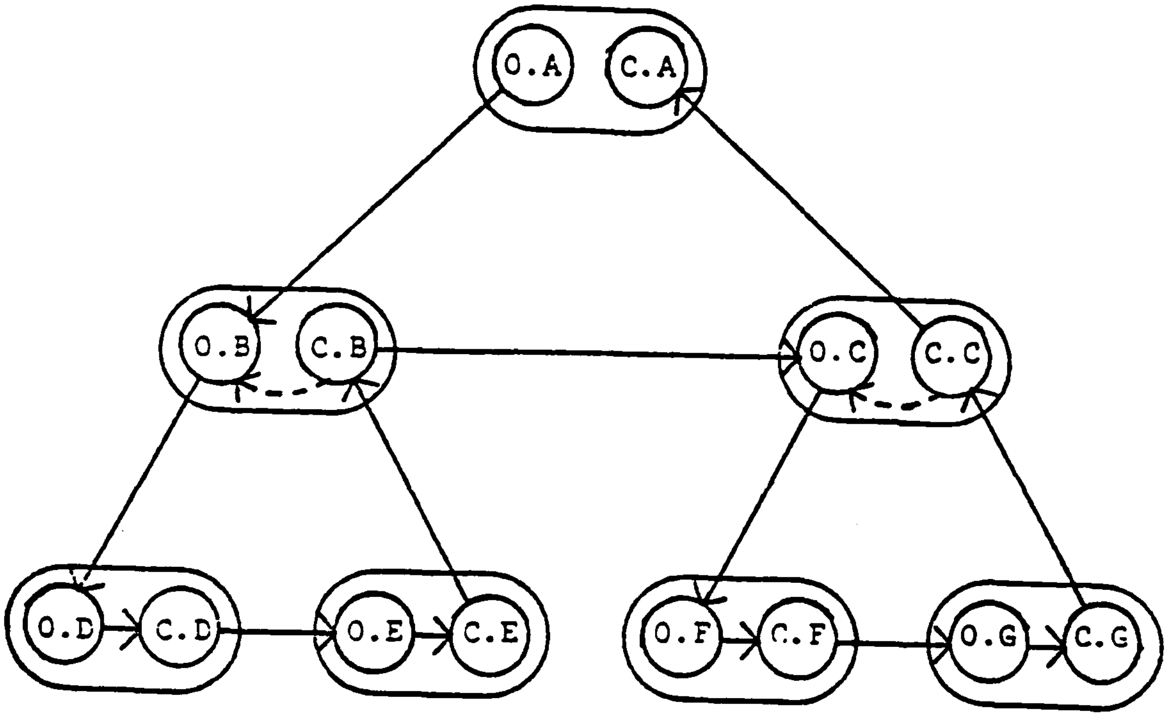
Finally we will put control variables into the dictionary. For each type of qualifier keyword, we find from the program specification all the qualified names with that qualifier. Next we search the dictionary for the suffix name. If the suffix is a declared data entity, the full name of the control variable is formed from the full name of the associated data entity. Otherwise, the qualified name is an unrecognizable symbol and is reported as such to the user.

4.4 CREATION OF ARRAY GRAPH

4.4.1 ENTER HIERARCHICAL RELATIONSHIPS (ENHRREL)

The data stored in external sequential files are simply a string of bits. The use of data description statements allows the user to treat them as structured. Therefore, the system has to transform the data files from a linear form to the structured form which is described by the user. For this purpose, we envisage that there are two program events corresponding to each data entity, one for opening the data and the other for closing the data. The sequential order of data in the external file requires these opening and closing events be arranged in a strict order. The precedence relationship among these program events can be established as follows. If a data entity contains some members, then its opening event precedes the opening event of its first member and its closing event follows the closing event of its last member. In addition, the closing event of its nth member precedes the opening event of its n+1th member. In the case that a data entity is repeating, then the closing event of its n-1th instance precedes the opening event of its nth instance. Fig. 4.3 shows the precedence relationship of a sequential file. Because the data node B is repeating, there is an edge from the n-1th instance of the closing event of node B to the nth instance of the opening event of node B. The edge is shown as a dashed line. The existence of this feedback edge causes a cycle in the Array Graph and this cycle ensures us that the reading of an instance of the field D will be followed by the reading of an instance of E. It should be noted that the subscript expression associated with the edge from the event C.B to the event O.B is of the form I-1 which allows us to remove it and break the cycle during the scheduling phase.

```
A IS FILE (B(*),C(*)) ;
B IS RECORD (D,E) ;
C IS RECORD (F,G) ;
D,E,F,G ARE FIELD ;
```



* O.X: opening event for data X

* C.X: closing event for data X

Fig. 4.3 Precedence relationship of a data structure

We envisage that for each field entity there is a third node which corresponds to the available event of the data. The opening event of an input field must precede its available event, and the closing event of an output field should follow its available event.

This view assures us that we can always read the input files sequentially and store them in the main memory before any computation starts. If there are variable structures, i.e., structures of varying field length or varying number of repetitions, then we may have to include some assertions in the reading process. Afterwards we can do all the computation internally conforming with the constraint of data dependency which is implied by the assertions. At the end, all the fields in the output files are available and the informations for controlling the variable structure are available, too. We then take the data from main memory, assemble them into records, and write the records sequentially.

Actually we have in the Array Graph only one node, instead of the open, close, and available nodes mentioned above, for each data entity, as this helps compiler efficiency. For input files, we can view the nodes as corresponding to the opening events. For output files, the nodes corresponding to the closing events. The records stored in a sequential file have to be accessed in a strict order. Therefore, there is a precedence relationships among the data entities of an input or output file to assure that the records are accessed in the proper order. On the other hand, a record is composed of fields. The <u>membership relation</u> between a record and its constituent fields implies a precedence relationship, i.e. no field in an input record will be available until the record is read in. Similarly all the fields in an output record should be available before the record can be written out.

We will use the following definitions in discussing tree structures.

<u>Definition</u> For a data entity G, SON1(G) denotes its leftmost son.

<u>Definition</u> For a data entity G, RSON(G) denotes its rightmost son.

<u>Definition</u> For a data entity G, CEB(G) denotes the closest elder brother of G, i.e. the data entity which is to the immediate left of G among all the brothers of G.

<u>Definition</u> For a data entity G, CYB(G) denotes its closest younger brother, i.e. the data entity which is to the immediate right of G among all the brothers of G.

<u>Definition</u> For any tree with node G as the root, RDM(G) denotes the rightmost node on the frontier of the tree.

<u>Definition</u> For any tree with node G as the root, LDM(G) denotes the leftmost node on the frontier of the tree.

The precedence relationships in different file types is discussed in the following.

1) Input sequential file. Since the records in a sequential file are read in one at a time, the precedence relationship needs to assure that the records are read in the order they are present in the input file. A record may be composed of many fields. Therefore, after a record is read, it should be unpacked to get all the fields. If the records in a file are not unpacked in the order they are read, then we will need memory space to store the records. Therefore, it is advantageous to unpack the records when they are read in. This implies that all the fields in a sequential file will become available in the order they occur in the external file. Three kind of edges are drawn among the data nodes in an input sequential file.

a) Assume that a data node G is n dimensional. If SON1(G) exists and is m dimensional where m may be either n or n+1, then the following edge is drawn.

   SON1(G)(J1,...,Jm) <-1a- G(J1,...,Jn)

b) Assume that a data node G is n dimensional and FATHER(G) is k dimensional where k may be either n-1 or n depending on whether node G repeats or not. If CEB(G) exists and RDM(CEB(G)) is m dimensional, then the following edge is drawn.

   G(J1,...,Jn) <-1b- RDM(CEB(G))(J1,...,Jk,*,...,*)

c) Assuming that a data node G is n dimensional. If it is repeating, then the following edge is drawn.

   $G(J1,...,J_n)$ <-1c- $RDM(G)(J1,...,J_n -1,*,...,*)$

If a data node in an input sequential file corresponds to the opening event of that data, we can interpret the above edges in the following way. The edges of type 1a say that a higher level data instance should be ready before all of the data instances corresponding to the first member of it can be read. The edges of type 1b say that all the brothers within the same instance of their father should be read in the order they are declared in the data structure. The edges of type 1c say that if a data node is repeating, then one instance of it is not ready to be read until the last field in the previous instance of it is read.

2) Output sequential file. The records of an output sequential file should be written out in a strict order. There may be several fields in a record, therefore, we may have to pack the fields before writing. Packing the fields when they become available is convenient for the code generation but poses extra restrictions on scheduling the assertions. For example, suppose a record node R contains three fields A, B, and C. If we insist that fields A, B, and C should be available in that order, the user would not be able to define the value of A in terms of C. Therefore, at or above the record level the precedence relationship requires that the records be written in strict order but below record level the precedence relationship will only require that the constituent fields of a record are ready before the record is written. Therefore, fields in a record do not have to be computed in the order they are packed into the record.

Three kinds of edges are drawn among the data entities above and including the record level of an output sequential file.

a) Assuming that G is an n dimensional data entity above the record level and RSON(G) , i.e. the rightmost son of G, is m dimensional. The following edge is drawn from RSON(G) to G.

   G(J1,...,Jn) <-2a- RSON(G)(J1,...,Jn,*)

b) If node G has a younger brother, then an edge will be drawn from node G to LDM(CYB(G)). Let G be an n dimensional node, FATHER(G) be a k dimensional node, and LDM(CYB(G)) be a m dimensional node. The edge to be drawn is as follows.

   LDM(CYB(G))(J1,...,Jk,...,Jm) <-2b- G(J1,...,Jk,*)

c) If node G is repeating, then the following edge is drawn from G to LDM(G). Let G be an n dimensional node and LDM(G) be a m dimensional node.

LDM(G)(J1,...,Jn,...Jm) <-2c- G(J1,...,Jn-1)

If we imagine that a data node in an output sequential file corresponds to the closing event of that data, then the edges mentioned above have the following interpretation. An edge of type 2a says that a data instance can be written out only after all the data instances corresponding to its last son are written out. An edge of type 2b says that all the instances of an elder brother within the same father instance should be written before any instance of its younger brother can be written. An edge of type 2c says that if a data node is repeating, then an instance of it cannot begin to be written until the previous instance is completely written.

Below the record level in an output file, the precedence relationships assures that a record will not be written out until all of its constituent fields are available. However, the relative order in which the fields are computed is not restricted. We will simply draw edges from all the descendants of a record node to it. Fig. 4.4 illustrate the edges in an output sequential file.

```
A IS FILE (B(*),C(*)) ;
B IS RECORD (D,E) ;
C IS RECORD (F,G) ;
D,E,F,G ARE FIELD ;
```

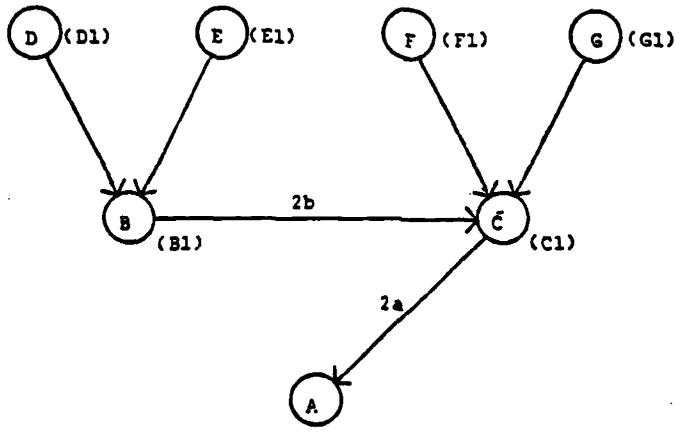


Fig. 4.4 The edges in an output sequential file

3) An input ISAM file. In an ISAM file, there is only one type of

record. The dimensionality of the record node IR is the same as that of the associated control variable POINTER.IR. Since the record instances are accessed with the keys, it is possible to read the records in the order of the keys. If the ISAM file is a pure source file to the program, the keys in the POINTER.IR array can be used in any order. On the other hand, if the ISAM file is used as a source and target file, the records should be processed in a sequential way, therefore, the keys in the POINTER array should be used sequentially to access the records. Below the record level, we can have the similar precedence relationship as in a SAM file because we may have to unpack the fields.

4) An output ISAM file. If an ISAM file is a pure target file, the output records will be added to the file. If it is a source and target file to the program, then only the selected records may be updated. In order to assure that each updated record includes the effects of previous updates, we will have to update and write out a record before the next record is read in. Therefore, the keys in the POINTER array should be used sequentially. However the fields in an output record can be computed in any order. Below record level the precedence relationships only reflect the membership of the fields within the record.

5) Interim variable. There are no I/O actions concerning interim variables. They are stored in main memory and referenced as fields. Therefore, there is no relative precedence relationship among the interim fields. But we still draw edges which reflect the membership among the data entities to facilitate *range propagation* (refer to Chapter 5). Since an interim variable is considered to be part of an output file except that it will not be written out, the edges are drawn from the descendants to the ancestors.

4.4.2 ENTER DEPENDENCY RELATIONSHIPS (ENEXDP)

Two types of assertions, namely simple assertion and conditional assertion, may be used to define the values of interim variables and output variables. The execution of an assertion depends on the availability of all of its source variables, and its execution makes the target variable available. This is because a data entity must be defined before it is referenced and a data entity becomes available after the assertion in which it is the target variable is executed.

Procedure ENEXDP examines all the assertions twice. In the first pass, it checks whether the target variable of an assertion defines a sublinear function and can be used as an indirect indexing vector or not. An indirect indexing array should be defined by an assertion of the following form.

```
  X(I) = IF I=1 THEN (0 | 1)
                ELSE (X(I-1) | X(I-1)+1) ;
```

During the second pass, it analyzes every assertion and enters the precedence relations caused by explicit data dependency into the Array

Graph. Given a simple assertion, the left hand side of it is scanned to find the target variable. Then the expression on the right hand side is scanned to find all the source variables. For a conditional assertion, the THEN parts, ELSE parts, and the conditional expression parts are scanned in that order to find all the source and the target variables. The source variables in a conditional assertion are found in the conditional expressions, the THEN parts, and the ELSE parts. For every source variable an edge is drawn from it to the assertion node. It should be noted that one assertion defines one target variable only and no more than one target variable can appear in a conditional assertion.

The edge from the source variable to the assertion is of EDGE_TYPE 3 and the edge from the assertion to the target variable is of EDGE_TYPE 7. The DIMDIF is the dimensionality difference of the target node and the source node of the edge. The types of the subscript expressions of a source variable are stored in the subscript expression list associated with the edge. It should be noted that the subscript expressions of the target variable define a mapping from the node subscripts of the target variable to the node subscripts of the assertion. Because the edge corresponding to the occurrence of the target variable is drawn from the assertion node to the target variable, instead of from the target variable to the assertion node, the mapping should be inverted to form the subscript expression list of the edge. In Fig. 4.5 the data dependency of an assertion is shown. Notice that there is a list of subscripts associated with every node in the graph. For example, variable A is a two dimensional array. Subscripts <A,1> and <A,2> correspond to the first and second dimension of array A. The edge leading from node A to a1 has a subscript expression list associated with it. The subscript expressions are ordered in the way they are used in the subscript variable A(I,J-1).

al: C(I,J) = A(I,J-1) + B(I,4) ;

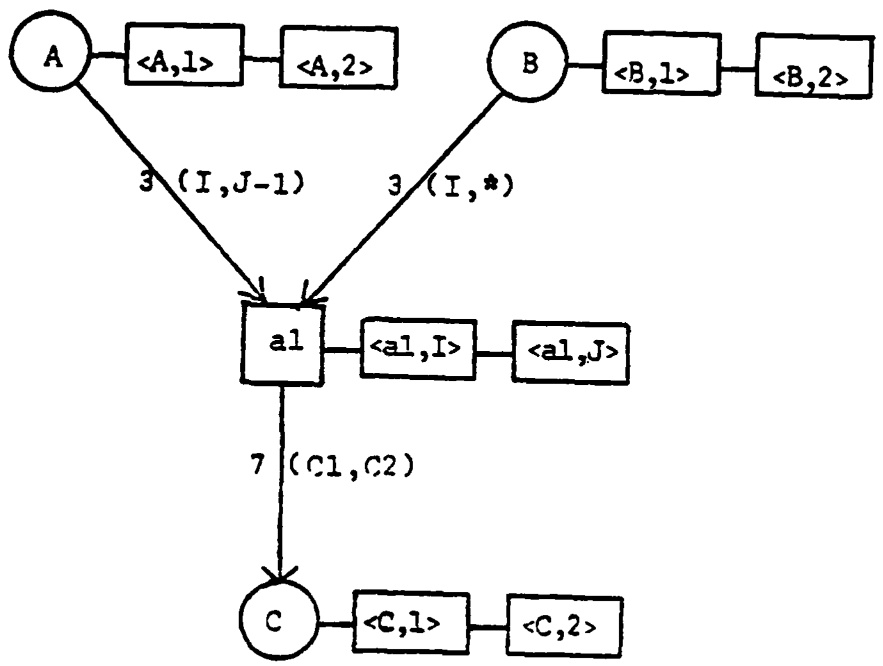


Fig. 4.5 The data dependency of an assertion

In addition to the explicit data dependency found in an assertion, there exists some implicit data dependency between the data entities and their associated control variables. Let TRGT denote the name of a data entity and NODE denote the name of the associated control variable which is composed of a keyword PREFIX followed by the name of the data entity.

1. If PREFIX = 'POINTER', then verify that TRGT is a keyed record and draw an edge.
   TRGT <-5- POINTER.TRGT, DIMDIF = 0 .
2. If PREFIX = 'SIZE', then verify that TRGT is repeating and draw an edge.
   TRGT(I) <-13- SIZE.TRGT, DIMDIF = 1 .
3. If PREFIX = 'END', then verify that TRGT is repeating and draw an edge.
   TRGT(I) <-14- END.TRGT(I-1), DIMDIF = 0 .
4. If PREFIX = 'FOUND', then varify that TRGT is a keyed record and draw an edge.
   FOUND.TRGT <-15- TRGT, DIMDIF = 0 .
5. If PREFIX = 'NEXT', then verify that TRGT is a field in an input sequential file and draw an edge.
   NEXT.TRGT <-16- TRGT, DIMDIF = 0 .
6. If PREFIX = 'SUBSET', then verify that TRGT is an output record. If it is an output record, then draw the following edge.
   TRGT <-17- SUBSET.TRGT, DIMDIF = 0 .
7. If PREFIX = 'LEN', then we draw an edge.
   TRGT <-20- LEN.TRGT, DIMDIF = 0 .

The subscript expression lists of these edges are for the moment empty. They will be constructed by the procedure FILLSUB later according to the EDGE_TYPE.

4.5 FINDING IMPLICIT PREDECESSORS (ENIMDP)

Many efforts have been made to make MODEL language tolerate some incompletenesses and inconsistencies in the specification. When incompletenesses and inconsistencies are found, warning messages or error messages are sent to the user. If practical, the MODEL processor tries to correct the specification in a reasonable way.

If an interim field is not defined by any assertion, an error message is sent to inform the user. It is probable that the user forgot to write the assertion. Therefore, the system should request an assertion from the user. However, if a field in a target file is not defined explicitly, the MODEL processor will try to find an implicit source to define that field. The MODEL processor tolerates this kind of incompleteness and saves the user work of writing assertions for merely copying fields from a source file to a target file.

Given a field in a target file which is not explicitly defined by any assertion, we will search for a field with the same name in another file according to the following order of priority. The idea is to make some reasonable assumption so that the undefined field will get a value.

Rule 1: If the undefined field is in a file which is both a source and target file, then the value in the corresponding field in the old record is taken as the value for it.

Rule 2: If Rule 1 does not apply, then the processor tries to find a same-named field in other source files. If one is found, it is assumed to be the source. If more than one is found, then the processor arbitrarily picks one as the source and prints a message to indicate that there was ambiguity.

Rule 3: If the above are unsuccessful, the processor tries to find a field with the same name in other output files. If one is found, it is taken as the source, and if more than one is found, then one is taken arbitrarily, with a corresponding message to the user regarding the ambiguity.

In the above cases where an implicit predecessor is found successfully, an assertion which defines the target variable by the implicit predecessor is generated as if it were entered by the user.

4.6 DIMENSION PROPAGATION (DIMPROP)

The source and the target variables in an assertion may be arrays. In order to reference an element of an N dimensional array, the user should subscript the array name with N subscript expressions. A subscriptless dialect of the MODEL language allows the user to omit subscripts in assertions in certain cases which do not lead to ambiguity. Therefore, the number of subscript expressions following an array variable does not necessarily indicate its actual dimensionality. Furthermore, the declaration of a multi-dimensional interim array may be simplified by omitting the data description statements for the higher level groups. The omission of subscript expressions in assertions and the omission of the higher level data description can be viewed as incompleteness or inconsistency of the specification. However, they are tolerated by the MODEL processor, and a process called dimension propagation is used to resolve inconsistencies of the dimensionality for the interim variables and missing subscripts in assertions.

All the nodes in input and output files should be declared precisely, using data description statements. Their number of dimensions can therefore be derived directly from the data description statements. Associated with every edge there is a field DIMDIF which denotes the dimension difference between the source and the target nodes of the edge. The number of dimensions of a node can be propagated along the edges of the Array Graph.

The dimension propagation algorithm is briefly described in the following. Let N denote the set of nodes in the Array Graph, array C store the current number of dimensions, and array D store the initially declared number of dimensions for each node in N. A queue Q keeps all the nodes whose calculated dimension could possibly be changed.

Algorithm 4.1 Dimension Propagation

Input. Array Graph.

Output. VIR_DIM: An attribute in the dictionary which contains the number of dimensions of a node.

```
1. For each node n in N, let C(n) be D(n) and put node n in Q.
2. If Q is empty, then exit.
3. Pick a node n from Q, remove it from Q. Let dim be 0.
4. For every incoming edge from node s to n, let dim be the maximum of
   dim and C(s)+DIMDIF.
5. For every outgoing edge from node n to t, let dim be the maximum of
   dim and C(t)-DIMDIF.
6. If dim<=C(n), go to step 2.
7. Else, the node n has a new updated dimension. Let C(n) be dim.
8. For every incoming edge from node s to n, append s to Q.
9. For every outgoing edge from node n to t, append t to Q.
10. If more than N*N nodes have been taken from the queue, then halt and
    issue an error message - there exists a propagation cycle.
```

If the process converges, then every node will have a finite dimension. However, it is possible that a cycle in the graph causes an endless increase in the dimensions. Consider for example the following specification.

```
(F, H) ARE FIELD ;
I IS SUBSCRIPT ;
IF I=1 THEN H(I) = 5 ; ELSE H(I) = F+1 ;
IF I=1 THEN F(I) = 6 ; ELSE F(I) = H+1 ;
```

The first assertion implies that the dimension of H is larger by 1 than that of F, i.e. C(H)>C(F). The second assertion states that C(F)>C(H). Applying our algorithm to this specification will result in endless loop of alternately incrementing C(H) and C(F). In this case the system will send out an error message indicating that the dimension propagation process is in an infinite cycle and also print out the nodes involved in the cycle.

4.7 FILLING MISSING SUBSCRIPTS IN ASSERTIONS (FILLSUB)

In the dimension propagation phase we have determined the number of dimensions of every node. If the number of dimensions of a node is larger than its apparent number of dimensions, it is necessary to add the respective subscript and data structures. This is performed in the following three tasks.

Task 1: Generate the node subscript list.

If the node X is a data node, its node subscript list is (displayed here from last to first):

(FOR_EACH.Ak, ... , FOR_EACH.A1)

where Ak, ..., A1 is the list of the repeating ancestors of X in a top down order. If X itself is repeating than A1 is equal to X.

If the node is an assertion node, then it has already been assigned a partial subscript list by ENEXDP. This is the list of apparent subscripts in the assertion, i.e. all the subscripts appearing either on the L.H.S. or the R.H.S. of the assertion. Let the assertion be of the form:

a1: A(Ik, ..., I1) = f(....) ;

Let the R.H.S. contains the subscripts J1, ..., Jm not appearing on the L.H.S. and hence assumed to be reduced. Then the partial list assigned to a1 is (Ik, ..., I1,Jm, ...,J1) and its apparent dimensionality is determined to be d=k+m. As a result of the dimension propagation process we may have recomputed a new dimensionality c for a1 where c>=d. This will cause n=c-d new subscripts to be added to the subscript list of a1 which now appears as:

($n, ..., $1,Ik,...I1,Jm,....,J1)

where $1, ..., $n are the name of the new subscripts.

Task 2: Fill in Missing Subscripts in the Assertions.

Consider an instance of a subscripted variable A(Ij, ..., I1) in an assertion. The calculated dimension VIR_DIM for array A yields a value d which should be greater or equal to j. If n=d-j>0 we should add n new system generated subscripts $1 to $n, modifying the instance into A($n, ..., $1,Ij, ..., I1). It should be noted that the new subscripts are always added on the leftmost dimensions of the array variables.

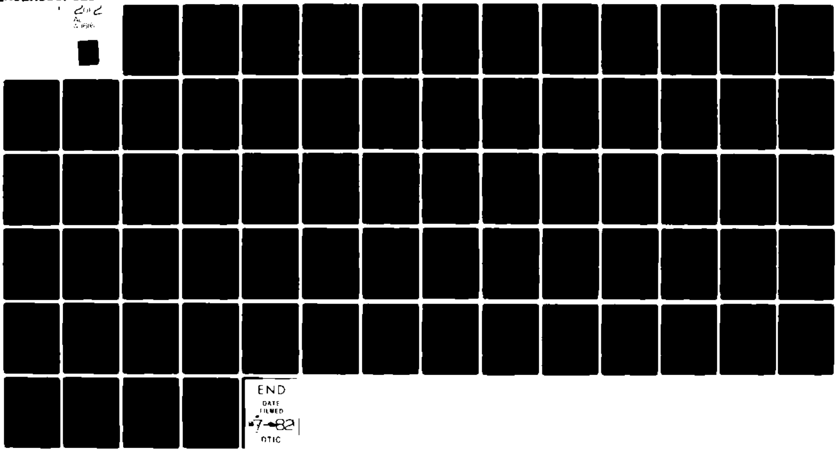
Task 3: Fill in the Subscript Expression List for the Edges.

All the edges except types 3 and 7 have been generated with an empty subscript expression list. Using the edge type and the dimensions of its source and target nodes, we generate a subscript expression list for each edge. Edges of type 3 and 7 have a partial subscript expression list based on their apparent appearance in the assertion. It may be necessary to expand this partial list. If n missing subscripts have been added to the variables in an assertion, then it is necessary to add n subscript expressions to the edges which correspond to the instances of the variables in the assertion.

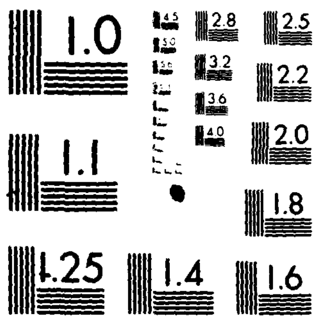
AD-A116 161 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
MODEL PROGRAM GENERATOR: SYSTEM AND PROGRAMMING DOCUMENTATION, --ETC(U)
MAY 82 K LU N00014-76-C-0416

UNCLASSIFIED NL

2 of 2



END
DATE
FILMED
7-82
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

CHAPTER 5

RANGE PROPAGATION

5.1 INTRODUCTION

The structures of variables are declared in data description statements. Every variable is considered an array of some dimensions. The number of elements in an array variable is determined by the dimensionality of the array and the sizes of each of the array dimensions. The size of an array dimension is called the range of that dimension. The range information allows us to allocate memory space for the array variables and generate iteration control statements which will define every element in the arrays. The use of subscripts in assertions makes it possible to define multiple elements of an array through one assertion. We can instantiate an assertion by fixing its subscript values. Then every instance of the assertion defines one single data element. The ranges of the assertion's subscripts restrict the number of instances of an assertion, which in turn defines the number of times that the assertion will be executed. The ranges of array dimensions and assertion subscripts are used in the later phases to synthesize the program.

Much information is not given explicitly in the specification. For instance users are allowed in assertions to use free subscripts for which the range is not specified. Also the range specifications of some array dimensions may be omitted. Therefore an algorithm is needed to derive ranges for certain assertion subscripts and array dimensions.

There is yet another reason why we want to analyze the subscript ranges. A criterion for placing a number of assertions in the scope of one loop is that they all have subscripts of the same range. From the point of view of program optimization it is preferred to have the loop scope as large as possible. It is important therefore to identify the subscripts of the same range. By propagating the specified range information to all the assertion subscripts and array dimensions we not only find the ranges which have been incompletely specified, but also identify the ranges which are equal.

5.2 LANGUAGE CONSTRUCTS FOR RANGE SPECIFICATION

A multi-dimensional array is declared as a hierarchical data structure with the most significant dimension specified at the top level. The range of a dimension may not depend on the subscript value of less significant dimension. The range of an array dimension may be specified in MODEL in several alternate ways as follows:

(1) Through a data description statement. A constant number of repetitions of a data structure may be specified in the data description statement which describes the parent structure.

(2) By defining the value of a SIZE qualified control variable (Refer to section 3.4.). For example, if group X repeats M times and M is a variable itself, we may use the following assertion to specify its range:

```
SIZE.X = M ;
```

A SIZE qualified variable is an interim variable of at most one dimension less than that of the suffix variable. Its value is used to define the range of the last dimension of the suffix variable (i.e. X). Consider an N dimensional repeating group X. Assume that the ranges of all its dimensions except the least significant one are defined elsewhere. By definition, SIZE.X is at most an N-1 dimensional array and the range of its dimensions is exactly the same as the range of corresponding dimensions of data structure X. Since the values in array SIZE.X can be different from one another, the array X may not have a regular (i.e. rectangular) shape, but have "jagged edges." This can be stated formally as follows:

$X(S_1, S_2, \ldots, S_k, \ldots, S_n)$ is in X iff

$SIZE.X(S_1, \ldots, S_k)$ is in SIZE.X &

$1 <= S_n <= SIZE.X(S_1, \ldots, S_k)$

(3) By defining the value of an END qualified control variable. The END array is of boolean type. It determines the range of the least significant dimension of the variable named in the suffix. Given an N dimensional array X, the associated control array END.X has the same structure as array X. The range of the Nth dimension is defined as the smallest positive integer Ln which satisfies the following conditions.

$END.X(S_1, \ldots, S_{n-1}, Ln) = TRUE$ &

$END.X(S_1, \ldots, S_{n-1}, S_n) = FALSE,$

for $1 <= S_n < Ln.$

(4) By using a subscript declaration statement to define a global subscript. The constant number of repetition can be specified in the statement. For example:

```
        I IS SUBSCRIPT (20) ;
```

(5) By system default. A repeating data structure which is a rightmost decendant and which is above or at the record level, may be assigned the end-of-file as its range if the user does not specify a range for it.

The mechanisms of SIZE and END arrays are not totally redundant. There are some essential differences between the SIZE and END arrays. First, the END array can define a minimum range of one, whereas the SIZE can define a range of zero. This is because the END array must have at least one value of boolean true. Secondly, the range specified by SIZE array is finite. But the range specified by END array may be infinite (through a user error in the range defining assertion, when there is no first boolean true condition). This is not checked by the system. Thirdly, the range specified by array SIZE.X(I1,..,Ik) may not depend on the array element X(I1,..,In), while END.X(I1,...,In) may depend on X(I1,...,In). For example, let X(1),...,X(k) be all the instances of an one dimensional array X whose range is specified by SIZE.X=k. In the program, the value of SIZE.X, i.e. k, must be computed before we compute any of the elements of X. If END control array is used, the range is specified by END.X(1), ... , END.X(k), and we only have to ensure that END.X(I-1) is computed before X(I) for 1<I<=k.

5.3 DEFINITIONS

Subscript variables belong to a special class of variables. While an ordinary variable can assume only a unique value, a subscript variable can take on a range of positive integer values. Subscript variables can be used as indices in array element references or in the same way as ordinary variables to compose complicated expressions. The meaning of subscripts is the same as their meaning in mathematical usage.

The following definitions are used in discussing subscripts.

Definition Let X be an N dimensional array represented in the Array Graph by a node. Let i be a positive integer. The tuple <X,i> is referred to as a node subscript. It denotes the ith dimension of the node of array X. Let a1 be an assertion node, and I a subscript variable referenced in the assertion a1. The tuple <a1,I> is referred to as a node subscript for I associated with the assertion node a1. If <n,d> is a node subscript, then R(<n,d>) denotes its range.

Node subscripts are grouped into range sets. Every range set contains the node subscripts which have the same range. However no two dimensions of the same node can be put into one range set even if they have the same ranges because every range set will later correspond to a level of nested loops in the generated program and no two dimensions of the same node can correspond to the same level of nesting loops.

Definition The range of a subscript that has been declared as a global subscript is the same in all assertions where it is used. There can only be one range associated with a global subscript.

Definition The range of a subscript that has not been declared as global is fixed within the scope of the assertion where it is used. It will be called a local subscript. A symbol used as a local subscript can have different ranges in different assertions.

There are two types of global subscripts in MODEL. One is specified by use of the qualifying keyword FOR_EACH in the prefix and a repeating data structure name in the suffix. The other is explicitly declared in a subscript declaration statement. (Refer to section 3.3.2.) The FOR_EACH type global subscript always has the range of the repeating data group named in the suffix associated with it. A user declared global subscript can have its range specified in the subscript declaration statement. By using global subscripts in assertions, the user can specify explicitly the range of assertion subscripts.

Local subscripts are all of the form SUBn where n is a positive integer. Users do not have to declare local subscripts (in subscript statement). The use of local subscripts in an assertion is like that of formal parameters in a function definition. They can be chosen arbitrarily within the scope of an assertion. This gives the user freedom to reuse the subscript names in different assertions.

5.4 DISCUSSION OF RANGE PROPAGATION

5.4.1 CRITERIA FOR RANGE PROPAGATION

In this section we discuss the conditions for propagating the range of a subscript from one node to another. A node subscript refers to either an array dimension or an assertion subscript. If two node subscripts are related through some dependency relation and one of them does not have an explicit range specification, we propagate the range from one to the other.

Let us consider first a simple assertion : B(I) = A(I) . Three entities are involved : the source variable A, the target variable B, and the assertion itself. All of them are one dimensional objects. The assertion states that the kth instance of the assertion corresponds to the kth instance of array B for all k in the range of B's dimension. There is a bijective mapping between the instances of the assertion and the instances of the array B. It is therefore very natural to believe that the range of the target variable B is the same as the range of the assertion. Additionally, from the subscript expression I in the term A(I) we can derive that the range of the assertion can be taken from the range of the array A. In short, whenever a simple subscript variable is used as a subscript expression it strongly suggests that we may propagate the range from one node subscript to another.

When a subscript expression of the form I-k is used in an assertion, where I is a subscript variable and k is a positive integer, there exists a one-to-one mapping between values of certain elements indexed by I and I-k. The mapping may be interpreted in two possible ways : assume the ranges of the arrays indexed with I and I-k subscripts are the same, or assume that the variable with the I-k subscript expression has k instances fewer than the variable with I subscript. We have decided to adopt the simpler assumption, that is, the ranges are the same. Therefore we will propagate ranges between the node subscripts indexed by subscript expression I and I-k.

It should be noted that we do not intend to modify or ignore a user specified range of a node subscript. The analysis mentioned above is used for two purposes. One is to derive a range for a node subscript which does not have an explicitly specified range. Second is to determine if it is possible to put two node subscripts into the same range set when both of them have user specified ranges and the ranges are the same. When two node subscripts have user specified ranges, we are interested in finding out whether their ranges are equal. Since there is no simple way to determine if two functions are equal in general, we will only check the assertions which define the range arrays by the other range array.

5.4.2 PRIORITY OF RANGE PROPAGATION

User specified ranges are associated with repeating data structures or declared global subscripts. The range specified for a data node is interpreted as the range of its least significant dimension. Ranges of node subscripts can be propagated along a path in the Array Graph from one node to another based on the following relations between respective node subscripts.

1. The two node subscripts are both global subscripts and have the same global subscript name.
2. One of the node subscripts corresponds to a dimension of a data node and the other corresponds to the same dimension number of the associated control variable.
3. The two node subscripts occur on the corresponding dimensions of two data nodes in the same data structure.
4. One node subscript is associated with an assertion node and the other is associated with a source variable of the assertion.
5. One node subscript is associated with an assertion node and the other is associated with the target variable of the assertion.

There may be several alternative paths (and directions) for propagating a range, and the range derived for a node subscript may depend on the choice of a path. The choice of path may also affect the efficiency of the generated program. Therefore, we will propagate ranges according to a priority order which attempts to obtain the highest efficiency. The priority order is as follows.

When a global subscript is used in several assertions, the ranges of the respective node subscripts (in these assertions) are the same. We may consider all the node subscripts with the same global subscript name as a group. Whenever any element in the group has its range defined, we will propagate the range to other elements in the same group. This type of propagation will have the top priority.

Next consider the data nodes and their associated control variables such as SIZE.X, END.X, POINTER.X, LEN.X, ..., etc. The dimensions of the control variables correspond to the dimensions of the variable named in the suffix from left to right. The corresponding dimensions of a data node and its associated control variables should have the same range. Similarly the corresponding dimensions of a data node and its higher level nodes in a data structure should have the same range.

If the range specification of local subscripts in assertions or array dimensions are not given explicitly, we will derive them by analyzing the respective subscript expressions in assertions. It is preferable to propagate the range from a target variable to an assertion rather than to propagate the range from a source variable to an assertion. Therefore, the range propagation between an assertion node and its target node or between a data node and its associated control variable will have the second priority.

Globally it is preferred to propagate the range from a variable in an output file backward to a variable in an input file than reversely. Thus we will assign the third priority to the propagation from an assertion node backward to its source variables and the fourth priority to the propagation from a data node forward to an assertion node in which it is referenced as a source variable.

<u>Example</u> Let array A be an input file with 20 elements, array C an output file with 10 elements and array B one dimensional interim array. The assertions

```
     a1:  B(I) = A(I) ;
     a2:  C(I) = B(I) ;
```

may lead us to assign either 20 or 10 as the range for array B, depending on the point of view taken. As far as the correctness is concerned, it does not make any difference whether 20 or 10 is used as the range of array B. But a smaller range would mean potentially less memory space and less computation time. Therefore the latter is more desirable. The range may be evaluated as follows. Since no global subscripts are used here, no propagation corresponding to the top priority can be achieved. The propagation from an assertion node to the target variable is second priority, therefore, the range of <C,1> and <B,1> should be propagated to <a2,I> and <a1,I> respectively. The range of subscript <B,1> will be that of <A,1> or <C,1> depends on whether we give higher priority to the propagation from <A,1> to <a1,I> or from <a2,I> to <B,1>. Since the latter has the higher priority, the range is propagated from array C all the way back to the assertion node a1. (Refer to Fig. 5.1.)

```
a1: B(I) = A(I) ;
a2: C(I) = B(I) ;
```

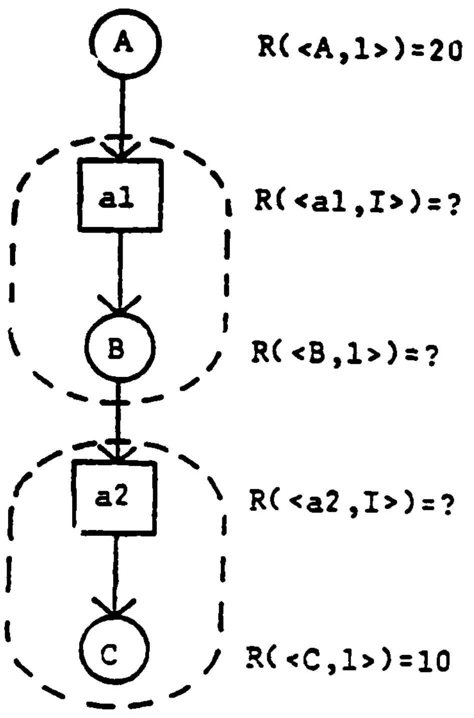


Fig. 5.1 Example of Range Propagation

In summary, we have divided the range propagation into four priority levels. The top level is based on use of global subscripts. The second level is based on the relation between data node and its associated control variables or between the assertions and their target variables. The third level is to propagate the range from an assertion backward to its source variables, and the fourth one is to propagate the range from a data array forward to the assertions in which it is referenced as a source variable.

5.4.3 REAL ARGUMENTS OF RANGE FUNCTIONS

Every node subscript will iterate over its range by a loop control statement in the generated program. A node in the Array Graph having N node subscripts associated with it will have an N level nested loop enclosing it. Every loop controls the iteration of a corresponding node subscript. We will show that the range specification of the node subscripts may have influence on the order that the loops can be nested and on the order of subscripts in referring to a range array.

When the ranges of the dimensions of an array are all constant, the array has a regular shape. We can access all of the array elements by iterating the subscripts in any order. For example, if we have a rectangular array A, we can access all of the array elements either row-wise or column-wise. However, if some of the dimension ranges of an array are specified by range arrays, it is no longer true that we can nest the loops in any order. In Fig. 5.2(a) two arrays A and B are both three dimensional arrays. The ranges of the third dimension of both arrays are specified by the SIZE.A array. In Fig. 5.2(b), a part of the flowchart for the specification in 5.2(a) is shown. The point is that the loop corresponds to node subscript <A,3> should be scheduled inside the loops of <A,1> and <A,2>. Because the loop control statement for <A,3> references the range array SIZE.A and the value of SIZE.A depends on the values of subscript <A,1> and <A,2>.

```
A IS FIELD;
B IS FIELD;
B(I,J,K) = A(I,J,K) ;
SIZE.A(I,J) = f(I,J) ;
```

Fig. 5.2(a) A range array with real arguments

```
  .
  .
  .
DO <A,1>;
  DO <A,2>;
    DO <A,3> = 1 TO SIZE.A(<A,1>,<A,2>);
      A(<A,1>,<A,2>,<A,3>);
      B(<A,1>,<A,2>,<A,3>) = A(<A,1>,<A,2>,<A,3>);
      B(<A,1>,<A,2>,<A,3>);
    END;
  END;
END;
  .
  .
  .
```

Fig. 5.2(b) Flowchart of 5.2(a)

A simple solution would be to require that the loops enclosing an array are nested according to the hierarchical order of the array dimensions. Thus, the dimension being declared on the top level of the data structure will be scheduled on the outmost level. Because the range of a dimension is not allowed to depend on the subscript value of any lower level dimension in the data structure, in the example above when the loop of <A,3> is to be scheduled, the loops of <A,1> and <A,2> would have been scheduled on the outer levels. However, this requirement is unnecessarily strong. For example, if we follow this scheme, then all the two dimensional arrays will have to be computed row-wise. With this restriction we may lose the opportunity to generate an optimal program.

A generalized solution would be to treat the range arrays as functions and find the real arguments of the range functions. For example, an N dimensional range array SIZE.X(I1,...,In) may be considered as a function which maps an N tuple of integers I1, ..., In to an integer value which is the range of the n+1th dimension of array X. Every subscript of the range array may be viewed as corresponding to an argument of the function. We will use the terms range array and <u>range function</u> interchangeably. Some of the function arguments may not affect the function value, namely the range does not vary with the value of these subscripts. The rest of the arguments which do play roles in determining the actual value are called <u>real arguments</u> of the range function.

By analyzing the assertion which defines a range array, we can find all the real arguments of the range array. If the range of a node subscript <n,d> is specified by a range array and the range array has some real arguments, the real arguments of the range array should correspond to some other node subscripts of node n. In the generated program the loops which correspond to the real arguments should be scheduled on the outside level of the loop which corresponds to the node subscript <n,d>. For example, consider the specification in Fig. 5.2(a). The range array SIZE.A has two real arguments, i.e. <SIZE.A,1> and <SIZE.A,2>. Since the node subscript <A,3> references the range array SIZE.A and the node subscripts <A,1> and <A,2> correspond to <SIZE.A,1> and <SIZE.A,2> respectively, node subscripts <A,1> and <A,2> will be stored in the <u>real argument list</u> of node subscript <A,3>. It is shown in Fig. 5.3. The loop iterated on <A,1> and <A,2> will be scheduled on the outside of the loop on <A,3>. Similarly, we can find the real argument lists for <a1,K> and <B,3>.

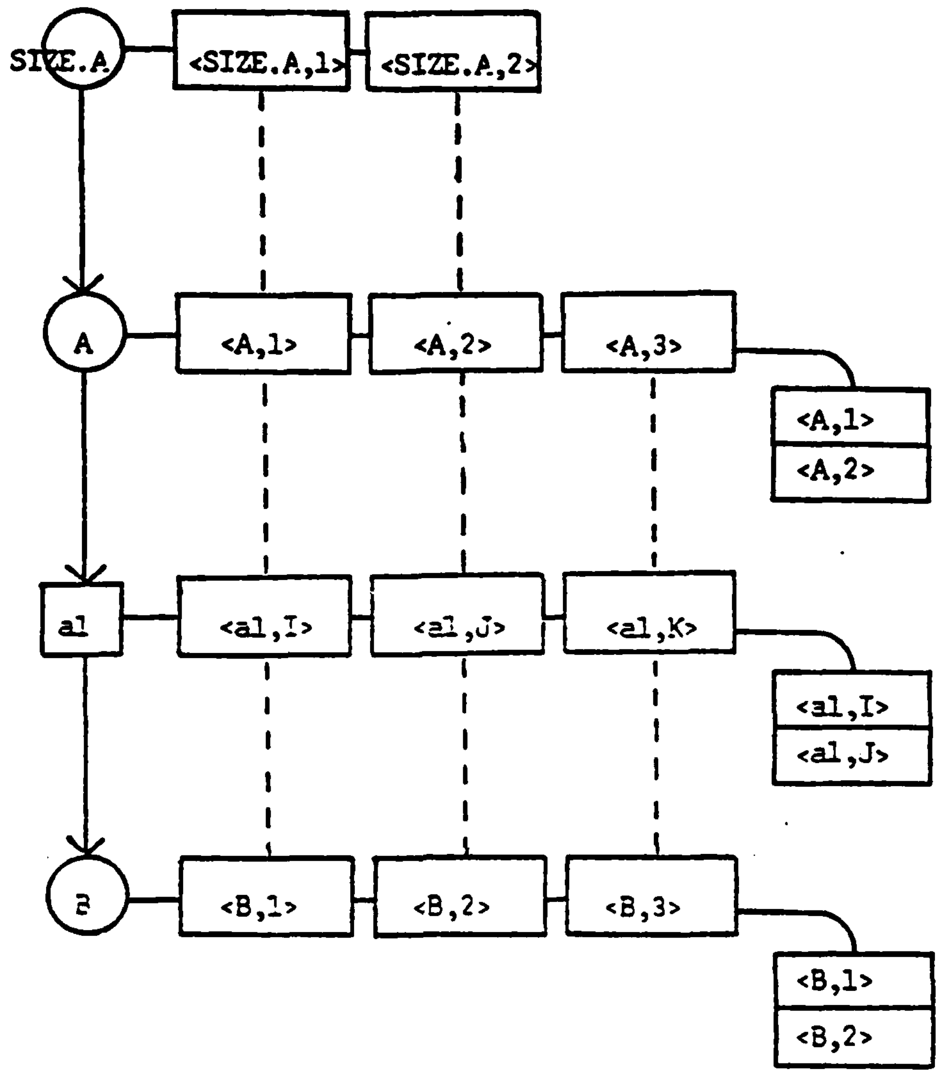


Fig. 5.3 Real argument lists of node subscripts

Example We will show how transposing an array effects the mapping between the real arguments of the range arrays. Let us examine the following assertions.

B(I,J,K) = A(J,I,K) ;

SIZE.A(M,N) = h(M,N) ;

Assuming that R(<A,1>) is equal to R(<B,2>) and R(<A,2>) is equal to R(<B,1>). The range for subscript <B,3> is obtained from R(<A,3>) which is given by SIZE.A. SIZE.B(N,M) should be equal to SIZE.A(M,N). All we need is a permutation of subscripts to make the range array SIZE.A the same as SIZE.B. A possible flowchart for the loops enclosing node A and B is shown in Fig. 5.4.

```
          .
          .
DO <A,1> ;
  DO <A,2> ;
    DO <A,3>= 1 TO SIZE.A(<A,1>,<A,2>) ;
      A(<A,1>,<A,2>,<A,3>) ;
    END ;
  END ;
END ;
          .
          .
DO <B,1> ;
  DO <B,2> ;
    DO <B,3>= 1 TO SIZE.A(<B,2>,<B,1>) ;
      B(<B,1>,<B,2>,<B,3>) ;
    END;
  END ;
END ;
          .
          .
```

Fig. 5.4 Transposition of real arguments of a range array

It should be noted that the order of the node subscripts <B,1> and <B,2> in the range array reference SIZE.A(<B,2>,<B,1>) is significant in the loop control statement for <B,3>. Therefore, in the real argument list associated with the node subscript <B,3> we should store the real arguments in the order of <B,2> followed by <B,1>. (Refer to Fig. 5.5)

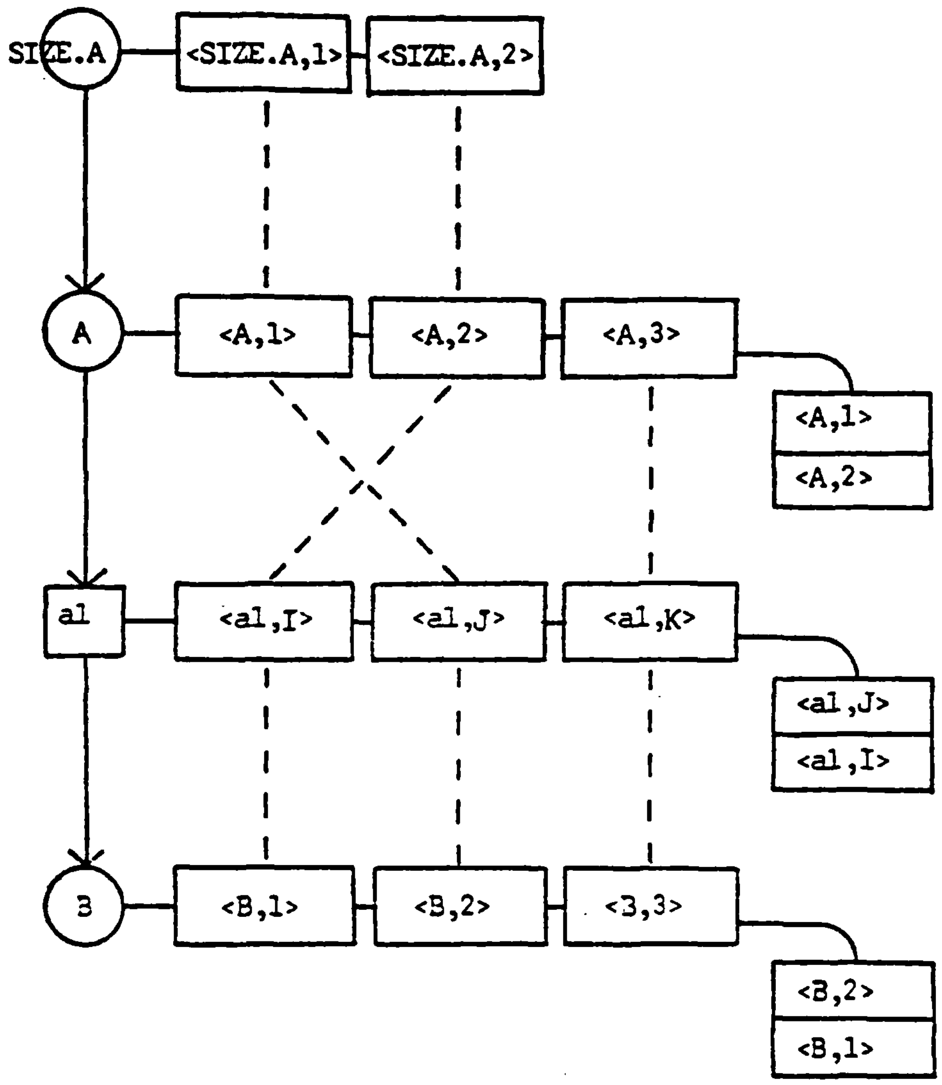


Fig. 5.5 The order of real arguments in the real argument list

5.5 RANGE PROPAGATION ALGORITHM (RNGPROP)

The range propagation algorithm consists of three steps. First of all, we locate the node subscripts which have user specified ranges (Algorithm 5.1). In the second step we propagate the explicit range specifications by partitioning the node subscript set into range sets (Algorithm 5.2). In the third step, we will propagate the real argument list(RAL) among the node subscripts in the same range set (Algorithm 5.3).

The data structure used are as follows. The total number of node subscripts is denoted by $ALLSUBS. Every node subscript is assigned a unique sequence number. A vector TERMC(DICTIND) of integer denotes the kind of range specification used for the least significant dimension of each node. It can have the values of 1-4 to denote the following conditions:

1: the data structure has a constant number of repetition.
2: the range is specified by an END array.
3: the range is specified by a SIZE array.
4: the range is implied by reading an end of file.

The vector LTERMC provides the same information for node subscripts as TERMC for the nodes. The contents of TERMC and LTERMC are computed by Algorithm 5.1.

<u>Algorithm 5.1 Find User Specified Ranges</u>

```
Output:
TERMC:  The type of user specified range of every node in the Array
        Graph.
LTERMC: The type of user specified range of every node subscript.
1. Initialize the vectors TERMC and LTERMC to 0.
2. For each node n, in turn do:
      If attribute VARYREP=0, then TERMC=1.
      If attribute ENDB>0, then TERMC=2.
      If attribute SIZEB>0, then TERMC=3.
3. For every node n, in turn do:
   If TERMC(n) is not equal zero, find the node subscript <n,d> which
   corresponds to the least significant dimension of node n. Set the
   LTERMC entry of the node subscript to TERMC(n).
```

Three arrays, HEADER, SETNEXT, and LRANGEP are used in step 2. Each of them has $ALLSUBS number of entries. HEADER(I) gives the sequence number of the header element of the block to which the Ith node subscript belongs. SETNEXT(I) links the Ith node subscript to the next node subscript in the same block, if any. When the Ith node subscript is the header of a block, then LRANGEP(I) shows the range of the Ith subscript. Algorithm 5.2 partitions the set of all the node subscripts. Initially every node subscript forms a block by itself. Then whenever we find that two node subscripts could have the same range and no range conflict would occur, we will merge their blocks. This merging process will continue until no further merging can be done. Since every node subscript can only be in one block at any moment, this is in fact a disjoint-set union problem[AHU 74]. The blocks formed in Algorithm 5.2 are called range sets.

<u>Algorithm 5.2 Propagation of Range Specification</u>

```
Input:
LTERMC: The type of user specified range for every node subscript.
Output:
RANGE:  A field in the LOCAL_SUB data structure of every node subscript.
        It contains the range set number where the node subscript
        belongs.
$RNGSET: The total number of range sets.
SET$RNG: The node number of the header of a range set.
Data structures:
$ALLSUBS: The total number of node subscripts.
HEADER($ALLSUBS): The node number of the header of the range set of a
        node subscript.
```

SETNEXT($ALLSUBS): For every node subscript, it points to the next node subscript of the same range set.

LRANGEP($ALLSUBS): If a node subscript is not the header of any range set, the value is -1. Else, if the node subscript has a user specified range, the value is the data node number of the range. Otherwise, the value is 0.

1. Initialization.
   Make every node subscript a block by itself. For all values of I from 1 to $ALLSUBS do:
   ```
   HEADER(I)=I,
   SETNEXT(I)=0, /* NO NEXT ELEMENT */
   LRANGEP(I)=node  of the range /* IF IT HAS A DEFINED RANGE */
             =0,  /* OTHERWISE              */
   ```
2. Merge blocks of the same global subscript name:
   For every node subscript with sequence number I, check whether it has a global subscript name. If it is a global subscript of the form FOR_EACH.X or user declared subscript X, let J be the sequence number of the node subscript which is associated with the least significant dimension of node X. Call procedure UNION(I,J) to merge the blocks containing these two subscripts.
3. Propagate ranges between data nodes and control arrays or target nodes and assertion nodes:
   For every edge in the Array Graph with edge type not equal to 3 check the type of the subscript expressions associated with the edge. These edges connect data arrays to the associated control arrays and the assertion nodes to their target variables. For every subscript of the source node, find the corresponding subscript in the target node. If the APR_MODE of the subscript expression is 1 or 2, merge them using procedure UNION.
4. Propagate ranges from assertion to source variable:
   Scan all the edges of type 3 which connect a source variable to an assertion. The range is to be propagated backwardly. If the subscript of the source node has a defined range, no merge will be done. Otherwise check if the APR_MODE of the subscript expression is 1 or 2. If yes, call procedure UNION to merge it with the corresponding subscript of the target node.
5. The same as step 4. Except that no merge will be done if the subscript of the target node has a defined range.
6. Check the header of each block. If it does not have a user defined range, check the elements of the block. If there exists an element which is associated with a data node at or above record level and being the rightmost node in an input file structure, we may use end-of-file as the default range.
7. Assign a <u>range</u> <u>set</u> <u>number</u> to every block of the partition. If a node subscript belongs to the kth block, put k into the RANGE field in the data structure LOCAL_SUB of the node subscript. Also store the node number which gives the range information of the block in SET$RNG(k) entry.

<u>Procedure</u> <u>UNION(I,J)</u>

Input:

I,J: The subscript sequence numbers of two node subscripts for which the range sets will be merged.

Output:

Modify the data structure HEADER, SETNEXT, and LRANGE to reflect the merging of the two range sets.

1. If both subscripts I and J are in the same block, exit.
2. If the blocks containing subscript I and J have different ranges, exit.
3. Put HEADER(I) into A.
4. Put HEADER(J) into B.
5. Change the HEADER entries of all the elements in the same block as J to A.
6. Append the list with the header B to the list with the header A.
7. Replace LRANGEP(A) by LRANGEP(B) if LRANGEP(A)=0.
8. Set LRANGEP(B) to -1.

Step three examines all the range sets. If the range of a range set is specified by a range array, a RAL is computed for every node subscript in the range set.

<u>Algorithm 5.3. Propagation of Real Argument List</u>

Input:

LTERMC: Type of user specified range of every node subscript.

RANGE: A field in the LOCAL_SUB data structure of every node subscript. It contains the range set number where the node subscript belongs.

Output:

RALP: A field in the data structure LOCAL_SUB of every node subscript. For every node subscript whose range is of types 2, 3, or 4, it points to a list of real arguments of the range function.

Data structure:

The real argument list pointed to by RALP consists of a list of elements which are stored in the data structure RAL. The fields in the RAL are as follows.

$RAL: The number of real arguments.

RSPOS($RAL): The subscript position of a real argument in the range array.

MSPOS($RAL): The subscript position of the corresponding real argument in the node subscript list.

1. For each node subscript which has a user specified range and the termination criterion is not constant, form the RAL for it and put it into a candidate queue. (Refer to Algorithm 5.4)
2. Iterate step 3 to step 7 until the candidate queue becomes empty.
3. Get a node subscript from the queue. Let it be the subscript S of node X. Propagate the RAL of S to other node subscripts in step 4, 5, 6, and 7. If any node subscript gets its RAL newly defined, put it into the candidate queue such that its RAL can be propagated to other subscripts.
4. For each outgoing edge from node X, propagate the RAL of subscript S from node X to the target node. (Refer to Algorithm 5.5)
5. For each incoming edge into node X, propagate the RAL of subscript S from node X back to the source node. (Refer to Algorithm 5.6)
6. If subscript S references a global subscript, propagate its RAL to the global subscript.
7. If subscript S is a global subscript, then propagate its RAL to all the subscripts which reference its name.
8. Stop.

<u>Algorithm 5.4. Find RAL from a range specifying assertion</u>

Suppose the range of the subscript <X,n> is specified by an assertion. Let the range array be SIZE.X or END.X. The algorithm tries to find the RAL for subscript <X,n>.

1. Put all the subscripts of the target variable of the assertion which defines the control variable SIZE.X or END.X into a list.
2. If the target variable is END.X, delete the subscript on its least significant dimension from the list.
3. Repeat for each of the subscripts in the RAL to check whether it is referenced on the right hand side. If yes, it is a Real Argument. Otherwise, delete it from the list.
4. The resulted list is the RAL of the subscript <X,n>.

<u>Algorithm 5.5. Propagation of RAL forward along an edge</u>

Assume S1 is a subscript of node X and there is an edge E from node X to node Y. The algorithm propagates the RAL of S1 to some subscript of node Y.

1. If the subscript expression of S1 is not type 1 or type 2, exit.
2. Let the corresponding subscript of node Y be S2. If RAL of S2 is defined, exit.
3. If the ranges of S1 and S2 are different, exit.
4. For each subscript in the RAL of S1, check its subscript expression type. If any one of them is not type 1, exit. Find their corresponding subscripts in node Y and form a new list. If the ranges of the corresponding subscripts are not the same, exit.
5. The newly formed subscript list is the RAL of S2.

<u>Algorithm 5.6. Propagation of RAL backward along an edge</u>

Assume S1 is a subscript of node X and there is an edge E from node Y to node X. The algorithm propagates the RAL of S1 to some subscript of node Y.

1. If there is no subscript of node Y corresponding to subscript S1, exit.
2. Let the corresponding subscript of node Y be S2. If RAL of S2 is defined, exit.
3. If the ranges of S1 and S2 are different, exit.
4. For every subscript Xi in the RAL of S1 find its corresponding subscript Yj of node Y.
   - 4.1 Let the subscript position of Xi in the local subscript list of node X be i.
   - 4.2 Check the LOCAL_SUB$ field in the data structure EDGE_SUBL associated with edge E. If the jth LOCAL_SUB$ is equal to i, the jth node subscript Yj in the local subscript list of node Y corresponds to Xi.
   - 4.3 Check the APR_MODE corresponding to subscript Yj in edge E. If it is not 1, exit.
   - 4.4 Check the RANGE field of the node subscript Yj and that of subscript Xi. If they are different, exit.
5. Form a subscript list which contains those subscripts Yj's of node Y. It is the RAL of subscript S2.

<u>Algorithm 5.7. Propagate RAL between Global subscripts</u>

Suppose subscript S1 of node X and subscript S2 of node Y have the same global subscript name. The algorithm propagates the RAL of S1 to S2.

1. If the RAL of S2 is defined, exit.
2. For each subscript T in the RAL of S1, get its range, say RT. Check

all the subscripts of node Y. If there is one and only one subscript U which has the same range as subscript T, then subscript U is the corresponding subscript of T. Otherwise, exit.

3. Form a subscript list which contains those subscripts U's of node Y. It is the RAL of S2.

5.6 DATA DEPENDENCY OF RANGE INFORMATION

In section 4.4.2 we have mentioned that range arrays cause implicit data dependency relationship. The edges of type 13 and 14 in the Array Graph represent this type of data dependency. However, it is not enough if we only have the edges from a range array SIZE.X or END.X to the node X. For every node in the Array Graph, no matter whether it is a data or an assertion node, as long as one of its node subscripts is in a range set where the range is deined by a range array, an edge should be drawn from the range array to that node.

We can tell the range of every node subscript only after the range propagation phase. Therefore, the correct time to add this type of data dependency relationship is after we have found all the range sets. If a range set has a range array as its range specification, then there will be edges emanating from the range array and terminating at every node in the range set. Subscript expressions of type 1 are associated with the edges emanating from a SIZE range array. Subscript expression of type 2 is associated with the least significant dimension of an END range array and type 1 subscript expressions are associated with the other dimensions of the END range array.

CHAPTER 6

SCHEDULING

6.1 OVERVIEW OF SCHEDULING

Through the phases of data dependency analysis, dimension propagation, and range propagation we have analyzed the user's specification and checked the consistency and completeness of the specification. In a non-procedural programming language, the execution sequence is not specified in the program specification. The objective in this chapter is to determine the order of execution in performing the specified computation. We have collected the needed information in the convenient form of the Array Graph. The Array Graph contains all the program activities as nodes and the data dependency relationships as edges. The next step toward constructing a program is ordering the program activities represented by the nodes of the Array Graph under the constraints posed by: a) the edges of the Array Graph, and b) considerations of computation efficiency. As stated in chapter 1, efficient scheduling is one of the main contributions of the reported research. This method of synthesizing the program is called scheduling here. It is followed by the actual program code generation.

Two rules which are frequently accepted in programming, except in cases where memory limitations are extremely severe, will be followed here as well. The first is that every input file is to be read only once. This rule will reduce the number of input activities which are usually relatively slow. If necessary we may store the input data in the memory for repetitive use. However, sometimes the memory price may be very high due to the large capacity of external storage. The second rule is that no values are to be recomputed. This means that once an element has been computed it will be retained as long as it is needed for later reference.

6.1.1 A BASIC APPROACH TO SCHEDULING

A correct but often inefficient realization of a computation can be obtained through the following scheduling method. Our eventual approach will be partly based on this simpler basic approach. The acyclic portions of an Array Graph may be scheduled very simply as follows. A topological sort algorithm can be applied to obtain a linear ordering of the nodes in the graph in accordance with the edge constraints. Multi-dimensional nodes are then enclosed within nested loop controls. Every loop iterates the respective node over the instances of one of the distinctive node subscripts of the node.

When there are cycles in the Array Graph, a topological sort will not succeed. Superficially, a cycle in the Array Graph means a circular definition which does not allow us to determine a linear order for the computation. Actually since the Array Graph masks some of the details of the relationships in the corresponding Underlying Graph (see Chapter 4), there may be a cycle in the Array Graph where there are no cycles in the corresponding Underlying Graph. Also iterative solution methods can be applied to perform the computations even where there are cycles in the Underlying Graph. We have to apply a deeper analysis of the nodes and subscript expressions used in assertions in the cycle. The cycles that are found to be really not circular can be resolved to generate a linear schedule. The method employed is briefly described as follows. The Array Graph is decomposed into subgraphs. Each subgraph is a most strongly connected component (MSCC). A MSCC in a directed graph is a maximal subgraph in which there is a path from any node to any other node. The deeper analysis is then applied to the MSCC components in the Array Graph. The analysis described in section 6.2 consists of search of a dimension that is common to all the nodes in the MSCC. If an edge is found in the MSCC which has an I-k type subscript expression associated with it, the edge may be deleted. This sometimes results in an acyclic subgraph which can be topologically sorted. If this method is not successful then other analysis methods, or alternatively an iterative solution method may be applied.

6.1.2 EFFICIENT SCHEDULING

In general, a schedule which satisfies the constraint of the data dependency relationship is not unique, if one exists. Therefore, there is a degree of freedom to select a schedule which meets efficiency requirements as well. We want to have a schedule with the fewest number of loops or with the least amount of working storage for the program variables. Although we will use here the results of the basic scheduling approach mentioned above, our method of scheduling consists essentially of a process of repeated merging of basic MSCCs in the Array Graph. As will be shown, in this way we can reduce the use of memory and computation time.

Non-procedural programming uses as many variables as the values that occur during the program computation. If we simply allocate separate memory space to each variable, as may be done in the basic approach, we will most probably get a program which uses a large amount of memory space and in some cases may not be executable. Therefore, we are here primarily concerned with memory efficiency of the program. Our approach is to examine the effect on use of memory due to merging of blocks of nodes of the same or related subscript ranges and form iteration loops for the selected subscripts enclosing the merged blocks. We will select mergers of blocks of nodes which reduces the use of memory the most.

In some cases we have an alternative of maximizing the scope of one loop at the cost of reducing the scope of one or more other loops. The choice of which loop scopes are maximized is based on comparison of memory requirements of the alternatives. The alternative that requires least memory space for program variables will be selected.

The repetitions indicated by the node subscripts are controlled by loop statements. The execution of loop statements takes some CPU time. If the loop scopes in a program are small, i.e. if they contain fewer nodes, then there will be more loops in the program and the overhead spent on the loop control statements will be increased. This is another reason why it is desirable to maximize the loop scopes in the generated programs.

6.1.3 OUTLINE OF THE CHAPTER

The material in sections 6.2, 6.3, and 6.4 forms a background to understanding the optimization in the scheduling algorithm. In section 6.2 we will discuss the analysis of MSCCs. The algorithm of our optimizing scheduler is based on deeper analysis of cycles. A similar approach was used previously in an earlier version of the MODEL processor. Some changes discovered in the course of the presently reported research have been added. The merger of components is discussed in section 6.3. There are two bases for merging of components: when components have the same subscript ranges and when they have related range (this is explained later). In section 6.4 we will introduce the memory penalty concept which will be used to evaluate the use of memory in a partially designed schedule. The memory penalty is the memory cost associated with a candidate subschedule. The scheduling algorithm is presented in section 6.5.

6.2 ANALYSIS OF MSCC

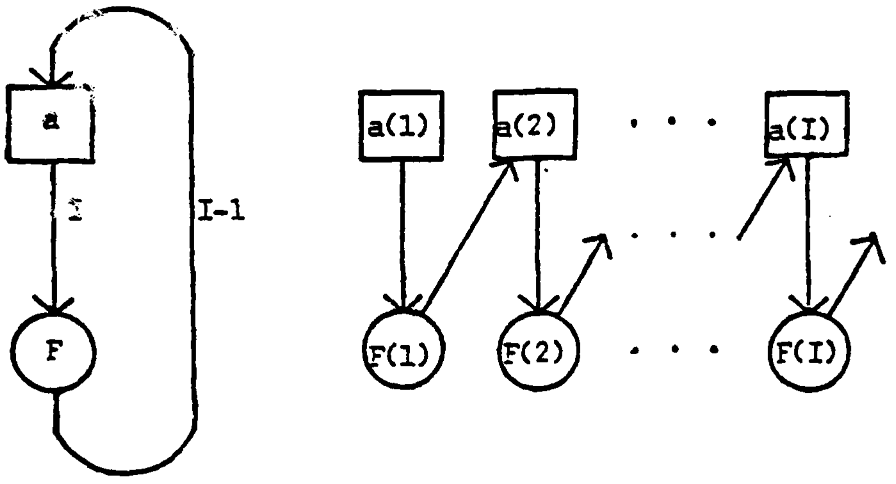
6.2.1 CYCLES IN THE ARRAY GRAPH

A cycle in the Array Graph means that a variable definition depends directly or indirectly on itself. An Array Graph is a compact representation of an Underlying Graph. It does not show the details of precedence relationships in the Underlying Graph. Therefore, the apparent circularity may be deceptive and not be reflected in the Underlying Graph. In this case a correct computation may be realized for an Array Graph cycle.

Consider for example the assertion in Fig. 6.1 which defines the factorial function. Because of the recursive definition there is a cycle in the Array Graph. But there is no cycle of precedence relationship in the corresponding Underlying Graph. Therefore, there exists a precedence ordered sequence for computing all the factorial values.

a(I): F(I) = IF I=1 THEN 1 ELSE I*F(I-1) ;

(a) Assertion



(b) Array Graph (c) Underlying Graph

Fig. 6.1 Example of cycles in the Array Graph

A MSCC in the Array Graph may or may not represent a circular definition. If it is not truly circular, we may be able to perform the respective computation by using an iteration loop. In section 6.2.2 we will discuss the conditions under which a MSCC can be enclosed in a loop. If these conditions are met, we will find the loop parameter to bracket the entire MSCC. Once such loop is found, since the loop indices are ascending, the precedence relationships between the respective loop instances is assured. Therefore, as shown in section 6.2.3 we delete edges with I-k subscript expressions and the MSCC may be decomposed. If the above method fails, there are other approaches to schedule a MSCC which will be discussed in section 6.2.4.

6.2.2 ENCLOSING A MSCC WITHIN A LOOP

The objective of iterative computations of a single data or an assertion node is to define all the elements corresponding to the values of node subscripts associated with the node. In general, the values of every node subscript can be stepped independently of other node subscript values. Therefore, a node with N node subscripts would have an N level nested loops enclosing it, and each level of the nested loop corresponds to one distinctive node subscript. We will associate with every loop a loop variable with values which are stepped up by one from one to the upper bound of a subscript range. All the nodes inside the scope of a loop will be executed once for every possible value of the loop variable. Generally if a node does not have a node subscript corresponding to a loop variable, the repetition would be redundant. We want to treat an entire MSCC in some manner as a single node, i.e. to compute all the elements of the nodes in the MSCC iteratively. We require however that all the nodes of a MSCC have a node subscript with which a loop brackets the MSCC. If one of the nodes does not have such a node subscript then the activity represented by the node, such as input/output, may be repeated, which will cause an erroneous computation. All the distinguished dimensions must then have the same range. It should be noted that the loop variable is stepped up each iteration by one, and no computation of a loop instance can depend on any computations in later loop instances.

Given a MSCC in the Array Graph, we will first check if all the nodes in the MSCC have more than zero dimensions. If every node does have at least one dimension to schedule, we will then check the subscript expressions on the edges of the MSCC to see if the entire MSCC can be enclosed within a loop. The edges in the Array Graph represent relationships between some elements of the nodes at the ends of the edges. The subscript expressions associated with edges reveal more precisely the precedence relationships between specific elements. In the following we examine the subscript expressions associated with an edge to determine if the nodes at the end of the edge can be scheduled within the scope of a loop.

<u>Definition</u> Let A be a node of n dimensions. Then <u>A</u> denotes the set of all the instances of node A, i.e. <u>A</u> = {A(I1,...,In)| 1<=Ik<=R(<A,k>), for 1<=k<=n }.

<u>Definition</u> Let A be a node of n dimensions. Then <u>A</u>(Ii=C1; Ij=C2; ...) denotes the set of all the instances of node A with the ith subscript Ii being C1 and the jth subscript Ij being C2, ... etc.

Consider an edge from node A(J1,...,Jm) to node B(I1,...,In) in the Array Graph:

B(I1,...,Ik,...,In) <—— A(E1,...,Ep,...,Em)

where J's and I's are the node subscripts of node A and B respectively, and E's are the subscripting expressions of A. Consider the subscript expressions of types 1, 2, 3, and 4.

1) If a subscript expression Ep is of type 1 and equals to Ik, then every element in B(Ik=c) depends only on the elements in A(Jp=c). Since B(Ik=c) does not depend on any element in A(Jp=d) with d>c, the Underlying Graph dependencies are satisfied if node A, followed by B, are bracketed by a loop where the parameters of the iteration are the pth dimension of A and the kth dimension of B. These are referred to as a distinguished dimension of A or of B.
2) If the subscript expression Ep is type 2 or 3 and equals to Ik-a, then for any positive integer c every element in B(Ik=c) depends only on the elements in A(Jp=c-a). Since the parameters of the bracketing loops are in ascending order (in step of 1) then this assures that A(Jp=d) is computed before B(Ik=c) with d<c. Thus it is allowed to schedule node A and B into one loop, with Ik and Jp the distinguished dimensions.
3) If the subscript expression Ep is type 4, then for any positive integers c and d every element in B(Ik=c) may depend on elements in A(Jp=d). We will be conservative and assume that every element in B(Ik=c) depends on at least one element in A(Jp=d) with d>c. Therefore, it is impossible to designate the pth dimension of A and the kth dimension of B as the distinguished dimensions for a loop.

<u>Example</u> Given an assertion a1 as follows. Let A and B be square arrays. There is an edge from array node A to assertion node a1.

```
a1(I,J):  B(I,J) = A(g,J);
                   where g is a type 4 subscript.
```

Consider the node set {A,a1}. Consider scheduling this set into one loop with <A,1> and <a1,I> as their distinguished dimensions. Let SA be {A(J1,J2)|J1=2} and SB be {a1(I,J)|I=1}. SB is in the first instance of the loop and SA is in the second instance of the loop, therefore SB precedes SA. Consider next the element a1(1,2) of SB. We can find an element A(2,2) in SA which precedes a1(1,2) because of the type 4 subscript on <A,1> dimension. SB and SA then precede each other, in the Underlying Graph, and therefore can not be scheduled.

<u>Example</u> Given the assertion a2 below.

```
a2(I,J):  Y(I,J) = X(I,J) + X(J,I);
```

X is a square array and subscripts <X,1>, <a2,I>, and <a2,J> have the same range. We want to schedule the node set {X,a2} in one loop with <X,1> and <a2,I> as the distinguished dimensions.
All the subscript expressions being used with node X are not type 4. However, in the term X(J,I) a subscript J occurs on the distinguished dimension of X, i.e. <X,1>. Since <a2,J> does not correspond to the distinguished dimension of node a2, it may be scheduled in an inner level loop and iterates faster than <a2,I>, therefore some array elements of X will be referenced before defined. Thus we should not form a loop with these designated distinguished dimensions.

From the examples above we know that the subscript expression on the distinguished dimension of a node must not be a general expression and it should correspond to the distinguished dimension of another node in the same loop, otherwise the loop can not be formed. Since the loop instances are strictly running upward starting from one and all the subscript expressions on the distinguished dimensions are of the form I or I-k, no reference goes to the later loop instances, therefore, no data dependency relationship is violated. In fact, by constructing the loop we have divided the whole computation into many smaller tasks where every task corresponds to a loop instance. It should be noticed that the formation of an outer loop does not exclude the possibility that the original computation involves an unsolvable cycle. What we are assured is that the outer loop divides the original problem into smaller ones and which can be solved easier.

6.2.3 DECOMPOSING A MSCC THROUGH DELETION OF EDGES

Consider now the case where an MSCC is scheduled in one loop based on the tests described in the previous subsection. The nodes in the MSCC have each a distinguished dimension which corresponds to the loop variable. Also the subscript expressions associated with the distinguished dimensions are of the form either I or I-k. We will show in the following that where the parameter of the loop is stepped up from one by a step of one then edges which have a subscript expression of type 2, i.e. I-k, are superfluous and can be removed.

Consider an edge of the form B(...,I,...) <—— A(...,I-k,...) where I-k and I occur on the pth and the qth dimension of nodes A and B, respectively. If node A and B are scheduled in the loop of I, then the elements in $\underline{A}$(Jp=I-k) have been evaluated in the I-kth loop instance and the elements in $\underline{B}$(Iq=I) are evaluated in the Ith loop instance. Since the values of loop variables are ascending, therefore every element of $\underline{A}$(Jp=I-k) precedes all the elements of $\underline{B}$(Iq=I). This implies that the precedence relation represented by the above edge is superflous as it is enforced by the order of evaluation of the respective elements. In short, when two nodes are scheduled in a loop of loop variable I, the precedence relationship presented by subscript expression I-k is subsumed by the order of loop execution. This is illustrated in Fig. 6.2, showing the Array Graph of a Factorial function which is defined with recursion. The recursion causes a cycle of two nodes {a1, FAC}.

a1: FAC(I) = IF I=1 THEN 1 ELSE I*FAC(I-1) ;

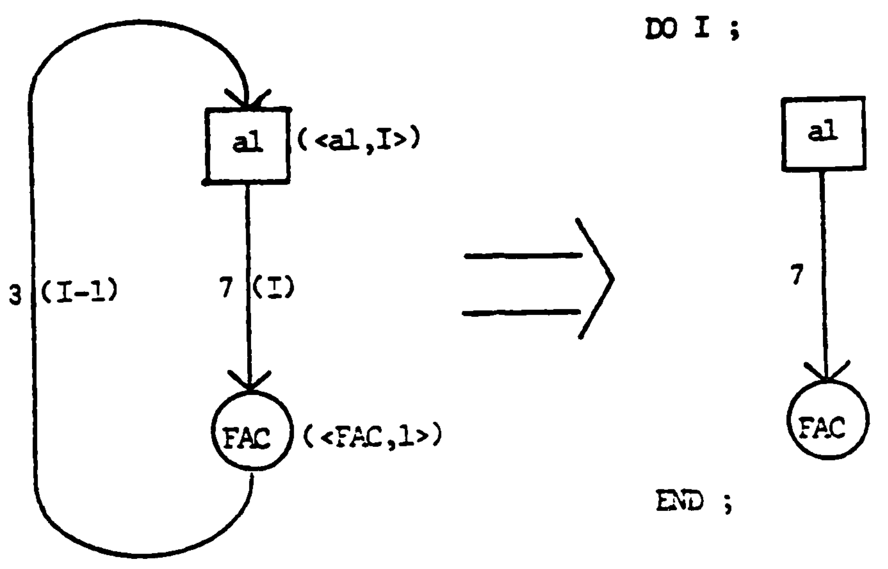


Fig. 6.2 Remove I-k edges in a loop

These two nodes can be scheduled in a loop iterating over node subscript <a1,I>. The kth instance of the assertion a1 is evaluated in the kth loop instance and it references the k-1th instance of the array FACT, which has been evaluated previously in the k-1th loop instance. Therefore the edge associated with subscript expression I-1 can be removed. There is no further a cycle in the Array Graph.

6.2.4 OTHER APPROACHES TO DECOMPOSING AN MSCC

There are a number of methods for scheduling a MSCC in an Array Graph. We have been primarily interested in the cases that a cycle can be implemented by a loop with the parameter that runs upward from one. However, not all the cycles can be implemented with this simple loop mechanism. Thus if the above approach fails it will be necessary to apply other methods. Consider first the case where the array elements may be evaluated in a sequence which does not follow the natural ascending order of subscripts. Consider for example the following specification which defines A, a vector of 50 elements.

Example
```
    A(I) = IF I=25 THEN X
           ELSE IF I<25 THEN A(I+2)+X
           ELSE A(I-1)+A(I-25) ;
```

A possible PL/I program to compute array A is as follows.

```
A(25) = X ;
DO I = 23 TO 1 BY -2 ;
  A(I) = A(I+2)+X ;
END ;
A(26) = A(25)+A(1) ;
DO I = 24 TO 2 BY -2 ;
  A(I) = A(I+2)+X ;
END ;
DO I = 27 TO 50 ;
  A(I) = A(I-1)+A(I-25) ;
END ;
```

When the subscript expressions are first order polynomials, we can divide an array nodes into many parts and compute the parts of the array separately [SHAS 78].

A cycle in the Array Graph may also be considered as a set of simultaneous equations and numerical methods such as Jacobi and Gauss-Seidel iterations can be applied to solve the system of equations [GREB 81]. Since splitting nodes in the Array Graph, as suggested by Shastry, is complicated to apply, the MSCCs which can not be decomposed may be treated similar to simultaneous equations and solved iteratively. In this dissertation we will refer only to the cases that a MSCC can be decomposed as described above. The other methods are described in the references.

6.2.5 A SIMPLE SCHEDULING ALGORITHM

The methods of scheduling an MSCC in a loop and attempting to decompose a MSCC may have to be applied repeatedly, depending on the outcome of each application. This section describes a simple scheduling algorithm which incorporates repeated application of the methods described earlier. It generates a correct schedule based on an Array Graph. However it does not include the consideration of program efficiency.

The algorithm consists of two mutually recursive procedures, SCHEDULE_GRAPH and SCHEDULE_COMPONENT. Given any Array Graph as input, SCHEDULE_GRAPH procedure finds the MSCCs in the Array Graph. The MSCCs are then sorted into a sequence {M1,M2,...,Mn} which retains the partial order of the precedence relationships between the MSCCs. SCHDULE_COMPONENT procedure then schedules each component separately. If Si is the schedule of component Mi, the sequence {S1,S2,...Sn} is returned as the schedule of the original graph.

The input to procedure SCHEDULE_COMPONENT is an MSCC, say Mi. If Mi is a single node component and there is no unscheduled node subscript associated with it, the node itself is returned as the schedule of the component. Otherwise, the component may be schedulable in a loop. The procedure tries to find a loop variable which satisfies the requirements discussed in the previous section. If a loop variable is found, say I, it then deletes the edges in component Mi with subscript expression I-k and marks the distinguished dimensions of the nodes in Mi as scheduled. Let Mi' denote the resulting graph. Then it calls the procedure SCHEDULE_GRAPH to produce a schedule for the graph Mi'. After SCHEDULE_GRAPH returns the schedule of Mi', a loop with loop variable I and loop body, the schedule of Mi' is formed by SCHEDULE_COMPONENT and returned as the schedule of Mi. If no loop variable can be found, SCHEDULE_COMPONENT sends a warning message to the user and calls the procedures described in section 6.2.4 to decompose the MSCC.

6.3 MERGER OF COMPONENTS TO ATTAIN HIGHER EFFICIENCY

The basic scheduling algorithm, described above, consists essentially of topological sorting of the nodes or MSCCs in the Array Graph and of the enclosing of these entities within the scope of nested loops for the respective dimensions. In contrast, the scheduling algorithm offered here considers the Array Graph globally and progressively merges components into the scope of a selected loop which reduces the most the use of memory and computing time. The scope of the loops in the schedule is thus progressively enlarged.

Given an Array Graph as input, we can construct a component graph where every MSCC is a component node and an edge is drawn from component A to component B if and only if there exists an edge in the original Array Graph which leads from a node in the component A to a node in the component B. The component graph is an acyclic graph. Note that the MSCCs in an Array Graph are not further divisible. The merger process starts with the MSCCs in the Array Graph as the *basic components*, and through merger it creates larger components progressively. A loop scope can be the union of some MSCCs. In this section we will discuss the merging of MSCCs in an Array Graph into the scope of one loop.

6.3.1 MERGER OF COMPONENTS WITH THE SAME RANGE

The condition for scheduling a set of component in one loop is that every component in the scope of a loop have a distinguished dimension corresponding to the loop variable. There are several condition on designating distinguished dimension of a node in an Array Graph or a Component Graph. First the distinguished dimensions of the components must be in the same range set and have a common range which specifies the number of iterations of the loop. The loop variable is stepped up by one in successive iterations. Therefore also the order of execution of elements of each component will be evaluated in this order. The second condition is that an evaluation of each instance of a component in a loop instance should not refer to values computed in later loop instances.

Further, components to be merged into the scope of a loop may not depend on any other component which does not have a distinguished dimension and which in turn depends on one of the components to be merged. The rule is that a set of components which can be scheduled in one loop should be equal to its <u>closure</u>. The closure of a set of components includes all the components which are reachable from any component in the set and which also reach any component in the set. For example, consider the component graph in Fig. 6.3. The components C1, C2, and C4 have a common dimension I. Still they can not be merged into the scope of a loop with the loop variable I. The closure of the set of components {C1, C2, C4} includes component C3. Since C3 does not iterate with subscript I, it can not be scheduled in the loop of I. Component C4 can be scheduled only after component C3. Therefore, at most we can merge components C1 and C2 or C2 and C4 into the scope of a loop.

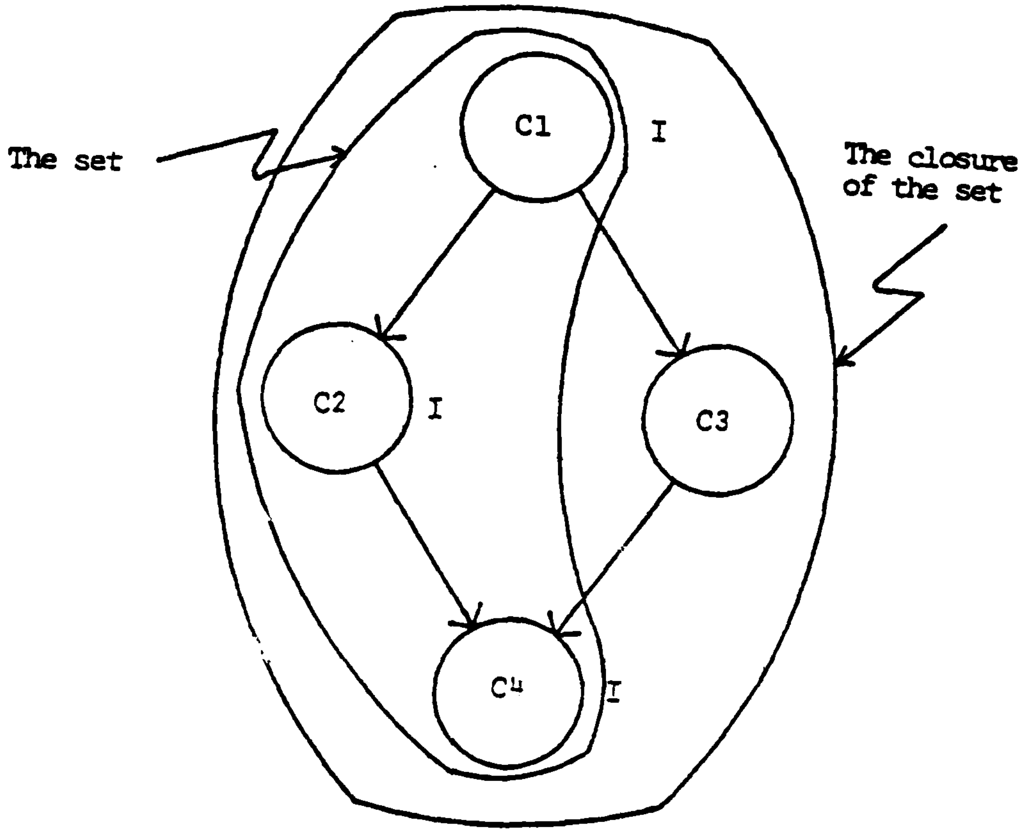


Fig. 6.3 Closure of a set of components

The search and selection of a distinguished dimension for each component in a set is similar to the analysis of subscript expressions in MSCCs described in section 6.2. We showed there that the subscript expressions associated with edges terminating at a component can not be type 4 and that subscript expressions associated with the edge should connect the distinguished dimensions of the components at the ends of the edge.

6.3.2 MERGER OF COMPONENTS WITH SUBLINEARLY RELATED RANGE

In the previous subsection, we considered merging components with distinguished dimensions which have exactly the same range as the loop variable. Every node is then executed once in each loop instance.

There is a large class of cases where subscript expressions are explicitly related, i.e. where we use an indirect subscript X(I) and X is a function of I. Statements with such an indirect subscript may in some case be conditionally executed in the scope of a loop for the parameter I. We will require that the indirect subscript expression X(I) have values which grow monotonically and slower than that of the loop variable I. This feature of sublinearity was already mentioned in section 4.4.2. As explained in [PNPR 80], use of indirect sublinear subscript is important in many instances, such as selecting a subset of records from a sequential file or merging two sequential files into one.

In section 4.4.2 we have discussed the criterion for recognizing a vector which can be used for indirect indexing. The values of elements of an indirect indexing vector grow slower than the subscript value of the elements. The range of its dimension will be called here the major range, while the range of its content will be called subrange relative to the major range. For example, the variable X in Fig. 6.4 satisfies these criteria. X is used in the subscript expression of the first dimension of node A and therefore R(<X,1>) is a major range and R(<A,1>) is a subrange relative to R(<X,1>).

```
X(I) = If I=1 THEN 1
              ELSE IF <condition is true> THEN X(I-1)+1
                                          ELSE X(I-1) ;

B(I) = A(X(I)) ;
```

Fig. 6.4 Example of indirect sublinear indexing in subscript expression

A subrange relative to a major range may be the major range of some other subranges. Therefore, the sublinear relationship between the ranges may form a tree with the maximal major range at the root. We merge major ranges and subranges in a bottom up order. By progressively merging each subrange with the next level major range finally we will obtain a loop which iterates in the maximal major range, and where all of its subranges are nested inside the loop. Such merger of subranges may not always be possible. For example, if type 4 subscript expression is used in the distinguished dimensions of a component, the precedence relationship will prevent us from scheduling this component into the scope of a loop.

When a set of components with a subrange and a major range are merged into the scope of a loop, the major range will be used as the loop range and the value of elements of the indirect indexing vector will be checked to evaluate only the elements which are within the subrange. An instance of the subrange is executed for each stepping up by 1 of the indirect indexing vector. The computation of the indirect index should precede the computation of any node within the subrange. This introduces an additional precedence relationship.

We will treat subscript expressions of types 5, 6, and 7 similar to types 1, 2, and 3, respectively, in checking the consistency of subscript expressions of the distinguished dimensions as discussed in section 6.2.1. If a check of the subscript expressions of the distinguished dimensions fails, i.e. some type 4 subscript expressions are used or the subscript expressions do not connect distinguished dimensions of the components, we will treat these indirect subscript expressions of type 5, 6, and 7 as type 4. If the check succeeds, we will add edges in the Array Graph from the indirect indexing vector to the nodes referencing it. This is similar to the addition of edges from a range array to the nodes referencing the range array.

6.4 MEMORY EFFICIENCY

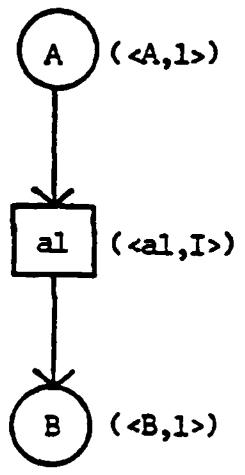
In some cases the same memory space may be shared by a number of variables, thereby using memory storage more efficiently. Small savings of memory space are not worth the cost of the analysis. For example, sharing memory space among few scalar variables does not save much memory space. Our approach will concentrate on having elements of the same array share the memory space. Since the range of each array dimension is in general large and there are several dimensions, the saving should be considerable. It should also be noted that memory space is statically allocated to the variables in the produced program. Compared with dynamic memory allocation, static memory allocation has the advantages of simplifying the program control in that there is no need to allocate memory space at run time. This also facilitates efficient random access of array elements.

Three alternative approaches to allocating memory are used:

1. <u>Physical Dimension</u>
   If all the elements along some array dimension have different memory spaces assigned to them, the memory space allocated is proportional to the range of the array dimension. This method of allocating memory will be referred to in the following as the <u>physical dimension</u>.
2. <u>Virtural Dimension</u>
   If all the elements along some array dimension share the same memory space, a single element memory space serves for the entire array dimension. We will refer to this method of allocation as <u>virtual dimension</u>.
3. <u>Window of width k</u>
   In some cases there is no need to store all the elements in an array dimension in main memory. But an array reference of the form

A(I-k) makes it necessary to keep k+1 array elements in main memory at any moment. This type of memory allocation will be referred to as <u>window of width k+1</u>.

For every array dimension we have to decide how the memory space is to be allocated. The memory allocation decision is related to the program execution sequence. Different program schedules may require different memory allocation approaches. For example, Fig. 6.5 shows two different schedules for copying a file. The one which reads all the records into the main memory then writes them out takes more memory space than the other one which copies the file, record by record.



Schedule-1

```
DO I ;
  READ(A(I)) ;
END ;

DO I ;
  B(I) = A(I) ;
END ;

DO I ;
  WRITE(B(I)) ;
END ;
```

Schedule-2

```
DO I ;
  READ(A(I)) ;
  B(I) = A(I) ;
  WRITE(B(I)) ;
END ;
```

Fig. 6.5 Two schedules for copying a file

In the following we will show how the memory allocation decisions are influenced by the program schedule and how the memory space requirement for the program variables is evaluated.

6.4.1 EVALUATION OF MEMORY USAGE

We will first consider in what units we should allocate memory space. If a data structure or substructure is used as an argument of a function or an operation, the whole structure must be passed between program modules. The relative position of its constituent elements becomes important to the computation. Therefore we can not allocate memory space to its elements separately. On the other hand, economic allocation of memory space requires that the unit be as small as possible. We will require that all the operations operate on fields. Operations on higher level structure must be therefore transformed into operations on elementary data structure. The memory space will therefore be allocated in the unit of fields.

The array dimensions above the unit data structure will be considered as logical array dimensions for which there may not be corresponding physical dimensions in the allocated memory space. One of the three approaches mentioned above may be used to allocate memory space. Since a virtual dimension requires less memory space than a physical dimension, we would not physically allocate memory space to an array dimension unless it is necessary based on the logic of the specification. In the following we will discuss the conditions when an array dimension has to be physical or window of width k.

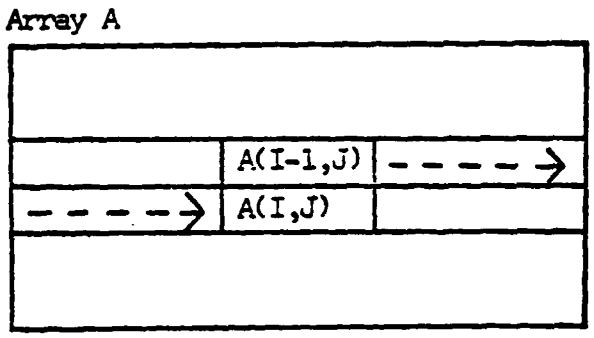
The values of data structures may be produced by some program activities such as reading an input file or evaluating an expression, and consumed by some other activities such as writing an output file or referencing an expression. If the production and consumption of the elements along an array dimension does not proceed in a planned order then all the array elements that are produced can not be discarded. All must be stored simultaneously in main memory.

Given a program schedule we can check whether the program activities which produce or consume the values along an array dimension are all in one loop. If not, that array dimension should be a physical dimension. If all the definitions and references of an array are in the same loop, we should further check whether any type 2 or 3 subscript expressions are used, because the occurrence of I-k type subscript implies the necessity of keeping previous k elements while computing a new array element. Thus the memory space for the array dimension should be a window of width k+1. It should be noted that if an array has its distinguished dimension using either a finite window or a physical dimension memory allocation scheme, all the loop for array dimensions which are scheduled nested inside the current loop have to be of physical dimensions. This is illustrated in Fig. 6.6, where a two dimensional array A is computed by a nested loop. Suppose the outer loop iterates over the first dimension of A, i.e. <A,1>. The presence of subscript expression I-1 requires a memory allocation scheme of window of width two for <A,1> dimension. Since the array element of A is computed row by row and the computation of array elements in one row depends on the value of array elements in the previous row, therefore, we will have to allocate two rows of memory space for array A.

```
a1:  A(I,J) = IF I=1 THEN f(J)
                       ELSE g(A(I-1),J) ;
```

(a) MODEL specification

```
DO I ;
  DO J ;
    a1(I,J) ;
  END ;
END ;
```



(b) Schedule (c) Memory requirement

Fig. 6.6 Effect of window dimension on the outer loop over dimensions on the inner loops

After the memory allocation approach for every array dimension has been determined, we can estimate the memory space requirement, which will serve as a measure of the program quality. Given an N dimensional array A, we can define the required memory space M for a node subscript <A,i> as follows.

M(<A,i>)	= 1	if the ith dimension is virtual,
	= k	if using window of width k,
	= upper bound of R(<A,i>)	if physical.

If an array dimension is not physical, the upper bound of its range is not used in calculating the memory requirement. The upper bound is needed to estimate the memory space for a physical dimension. Sometimes the range of an array dimension is specified by an assertion and the upper bound is not known until run time. In that case we can only assume the upper bound is infinity unless the user has specified an upper bound of the range in the data description statements. The memory space for array A is the product of M(<A,i>)'s for all the dimensions of A. The total memory requirement of a program is the sum of the memory space used by every array variable.

6.4.2 MEMORY PENALTY

Analysis of the loop scope leads to the selection of the memory allocation scheme for the respective array dimension. The memory penalty of a loop is defined as the memory cost of the arrays included in the loop scope. The memory cost is the difference in memory requirements between the ideal case (virtual dimension) and the memory requirements if the loop is formed. In order to evaluate the memory penalty of a loop, we first find all the nodes whose memory allocation scheme is influenced by the construction of the considered loop.

Whenever an Array Graph edge crosses the loop boundary, a source or target node of the nodes in the loop will be outside of the loop. Either one of the two nodes may require using the physical memory allocation scheme. For example, if an edge from a data node to an assertion node crosses the loop boundary, (i.e. the data node is in the scope of the loop while the assertion node is outside), the data node is defined in one loop and referenced outside it. Therefore, its array dimensions have to be physical. Similarly if the edge crossing the loop boundary is from an assertion node to a data node, the dimension of the target node has to be physical.

Each node under consideration may fall into one of the following three categories and the memory penalty can be computed accordingly.

1. A physical dimension for a distinguished dimension. This category is recognized by the existence of an edge which crosses a loop boundary. The memory requirement in ideal case is taken as that of a virtual dimension. The memory requirement for a loop is computed by multiplying the upper bounds of all the unscheduled dimensions and the dimension that is considered for a loop. The difference is the penalty of the loop for this array.
2. A virtual dimension for the distinguished dimension. In this case the loop boundary is not crossed by edges and all the subscript expressions on its distinguished dimension are type 1 subscripts. The memory penalty for a virtual dimension should be zero.
3. A window of width k+1 for the distinguished dimension. Similar to the virtual dimension category. No edges would cross the loop boundary. However subscript expressions of the form I-k on its distinguished dimension are allowed. The other unscheduled dimensions are considered to be physical dimensions. The penalty is computed similar to the first category.

<u>Example</u> Consider the memory penalty of a loop shown in Fig. 6.7. The ranges of subscripts I and J are 10 and 20 respectively, and every data element occupies one unit of memory space. The memory requirements in ideal cases for node A, B, C, and D are 1, 1, 1, and 1 respectively. The memory requirements if the loop is formed will be 10, 40, 1, and 200 respectively. Arrays A and D have to be physical and the first dimension of array B needs a window of width 2. The memory penalty for this loop is the difference of 251 and 4, i.e. 247 units of memory space.

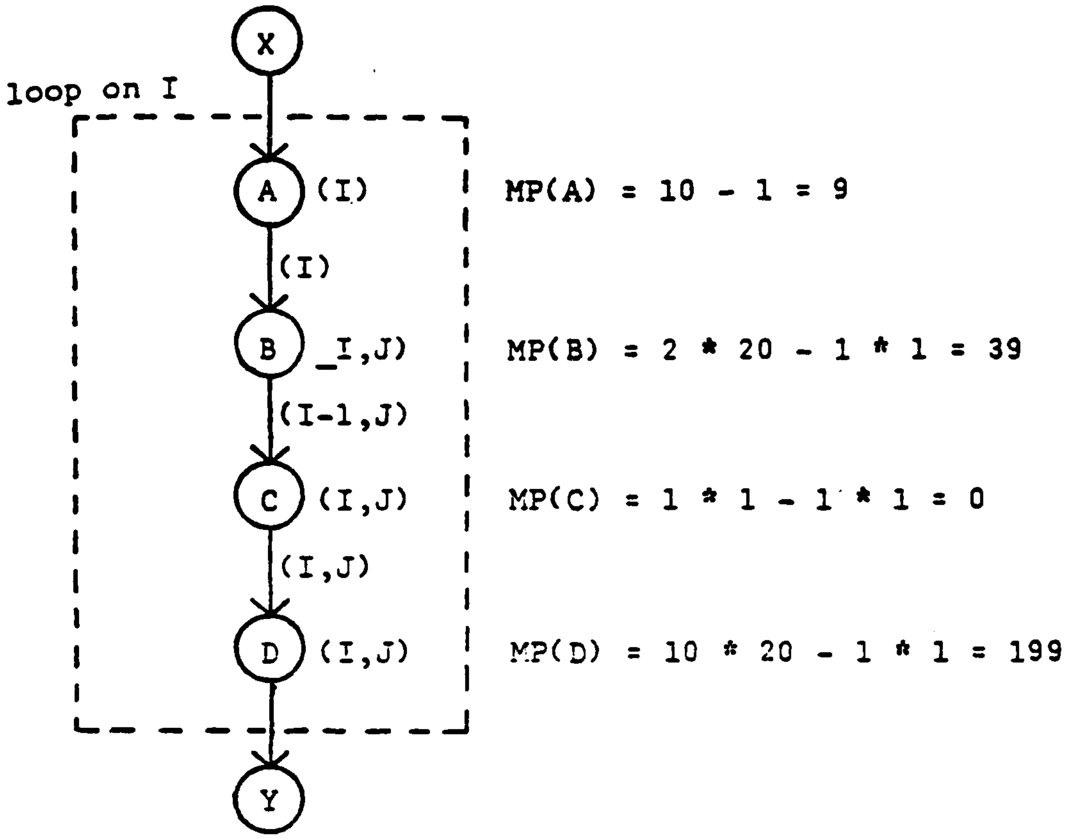

X
loop on I
A
(I)
(I)
B
(I,J)
(I-1,J)
C
(I,J)
(I,J)
D
(I,J)
Y
MP(A) = 10 - 1 = 9
MP(B) = 2 * 20 - 1 * 1 = 39
MP(C) = 1 * 1 - 1 * 1 = 0
MP(D) = 10 * 20 - 1 * 1 = 199

Fig. 6.7 Example of computing memory penalty

Information about the unscheduled dimensions may be used to compute the penalty more accurately. For example, some array dimensions must be physical dimensions because of the use of type 4 subscript expressions. During the process of scheduling, we can accumulate such information to speed up the memory penalty evaluations.

6.5 A HEURISTIC APPROACH TO MEMORY-EFFICIENT SCHEDULING

In general, there is a large number of schedules which can realize the computation of a program specification. The schedule with the minimal total memory requirement will be called an <u>absolute optimal program</u>. In principle it should be possible to enumerate all the possible schedules for an Array Graph, as there is a finite number of them, and then evaluate the memory requirement of each schedule. We would thus be able to find the absolute optimal schedule. For several reasons this method is not practical. The program events being scheduled are low level activities represented by nodes, i.e. statements and variables, and an Array Graph may easily consists of several hundred or even thousands of nodes. Also the nodes in the Array Graph may be multi-dimensional and the number of combinations of possible nested loops is very large. Further, the constraints on the feasible schedules are complicated. Thus enumerating all the feasible schedules would be prohibitive, and an exhaustive examination of all the feasible schedules to find the absolute optimum is not acceptable.

Instead we have adopted the heuristic approach as follows. Given an Array Graph as input, we first construct an acyclic component graph with the MSCCs in the Array Graph as nodes. Our objective is to repeatedly merge components in the component graph into blocks which correspond to loop scopes. This process will be applied repeatedly to the levels of nested loops. On the first application it will produce the outer level loops. The blocks are formed by merging as many components as possible which have the same or related ranges. The process is repeated for each lower level of the nested loops, based on the subgraph that corresponds to the higher level loop. This process may not result in the absolute optimal program as the outer level loop scopes are determined without the analysis of the effects of inner loop structures on the use of memory space. However considering the effect of inner loops on memory usage is a complex process and it represents a large increase in the number of alternatives that must be evaluated. The scope of the major loops in a program are maximized in our proposed approach and there is no, or little, effect of inner loops on memory usage. Thus this heuristic approach represents a good compromise between the amount of analysis involved and the payoff in reducing memory usage.

On each level of loops, the scheduling process consists of a trial scheduling for every range set in the corresponding Component Graph. A loop for the range R will enclose only the components which have dimensions in the range set associated with range R. The range sets related to R (through sublinear indirect indexes) will later be merged with the blocks of range R. The maximum loop scope for every range R is the range set of R.

The trial scheduling of each range set consists of finding the closure of the range set and an attempt to schedule nodes in the set which may be within the scope of the respective loop. We first merge into a block the components in the range set which do not have any predecessors in the closure of the range set. Progressively we will merge into the block other components which depend on those in the block, as far as possible. The merger involves selection of a distinguished dimension in each component, as described above. At the end we evaluate the memory penalty of the loop scope obtained by the trial scheduling. The loop with the smallest penalty will be scheduled finally. This process will be repeated with the unscheduled portion of the graph until all the components in the Component Graph are scheduled.

There are many possible orders for merging components in the closure of a range set, to form the scope of a loop. For example, we may arbitrarily pick a component in the middle of the Component Graph and merge it with its neighbor components or start with a component on which no other components depend and merge the components backward. However, considering all the possible orders of mergers will further increase the number of alternatives that must be evaluated. The order of mergers is unimportant in the case where the whole range set can be scheduled in one loop, i.e. it is the case that all the array dimensions may become virtual. No matter in what order we merge the components, we will finally get the same loop scope. Again, we selected the forward merging of the Component Graph as a good compromise between quality of the schedule and the amount of analysis.

It is necessary next to order the blocks associated with outside level loops in an execution sequence order. The memory cost will be the same for any order that maintains the precedence relations between these blocks. We choose to order the blocks by topological sorting. For every outer level loop we mark the distinguished dimensions of the blocks as scheduled.

We apply the scheduling algorithm recursively to each inner nested level loop by considering only the subgraph which contains the nodes in one loop scope. The resulting schedule will be the body of the outer level loop.

We will illustrate this process with an example of scheduling the Array Graph shown in Fig. 6.8. Every node is a MSCC by itself, and the initial Component Graph is in fact the Array Graph. The candidate ranges are R(<A,1>) and R(<B,1>). Assume that the repetition numbers are 500 and 200, respectively. The range set of R(<A,1>) contains three nodes: A, a1, and C. The closure of {A, a1, C} is itself. If we schedule the whole set into one loop, the penalty will be making array B physical. On the other hand, the trial scheduling of the range set of R(<B,1>) contains two nodes: B and a1. If this set is scehduled in one loop, the penalty will be making both array A and C physical. We will select the loop of R(<B,1>) since the size of array B is greater than the sum of the sizes of array A and C. We mark the component B and a1 as scheduled. There are two components left to be scheduled. We have no alternative but to schedule each of them in a separate loop. The resulting schedule is shown in Fig. 6.8(b).

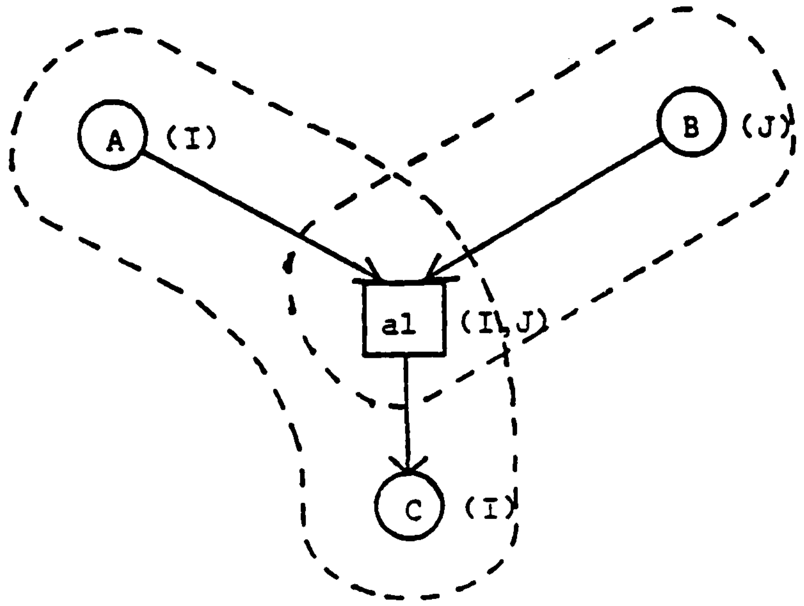
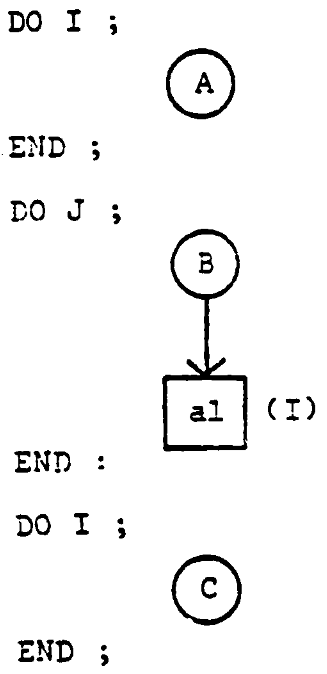


Fig. 6.8(a) An Array Graph to be scheduled



```
DO I ;
          A
END ;
DO J ;
          B
          a1  (I)
END :
DO I ;
          C
END ;
```

Fig. 6.8(b) The outer level loop structure

6.6 THE SCHEDULING ALGORITHM

The scheduling algorithm, called SCHEDULE, is documented below. The overall process is illustrated in Fig. 6.9. The solid lines show procedure calls and the dashed lines show passing of parameters and returns. The SCHEDULE process starts with construction of a reduced form of the Array Graph, which will be modified in the course of scheduling and is also easier to manipulate. It then calls a recursive procedure SCHEDULE_GRAPH. This procedure accepts an Array Graph as input and returns a schedule as output. SCHEDULE_GRAPH calls on a number of procedures to perform its tasks. It calls first the procedure STRONG to construct a Component Graph out of the reduced Array Graph (or subgraphs of it in recursive calls).

Next, the major iteration in SCHEDULE_GRAPH schedules the outer loop scopes. This iteration repeats until all the components in the Component Graph have been scheduled. This major iteration loop finds first all the candidate ranges.

Next there is a nested iteration for trial scheduling of all the candidates ranges. It consists of calls to four procedures. Procedure INDRSUB is called first to find the range sets of each candidate range. If a candidate range has some subranges related to it, the sets of the subranges will also be included in the major range set. CLOSURE is then called to get the subgraph for the closure of the range set. Then MAX_SCHED is called to do a trial scheduling. MAX_SCHED accepts as input a subgraph which consists of the closure of a respective range set and returns as output a loop scope which contains components in the closure of the range set that have been trial scheduled. The trial scheduling consists of repeated mergers into a loop scope of the components in the closure of the range set which do not depend on any other components. As a component is merged into the loop scope, it is deleted from the subgraph of closure of the range set. The merger repeats until no more components can be scheduled. Procedure EVALUATE is then called to compute the memory penalty associated with the loop scope.

At the end of the nested iterations for all the candidate ranges, SCHEDULE_GRAPH selects the loop scope with the smallest penalty. It will eventually form a part of the final schedule. The components in the selected loop scope are first merged into a single component and then marked off in the Component Graph.

The above major iteration loop is repeated, as noted above, until the Component Graph is empty. The outer loop scopes are thus all found. The corresponding components are topologically sorted. It is necessary then to find the nested loop scopes, if any, for each outer loop scope subgraph. As SCHEDULE_GRAPH selects the next component in the topological sorting, it calls the procedure EXTRACT to extract these subgraphs, which correspond to the selected loop scopes. Each of these subgraphs must be internally scheduled. EXTRACT calls SCHEDULE_GRAPH recursively, to schedule each of the subgraphs. A component that is not within a loop scope needs not be further internally scheduled.

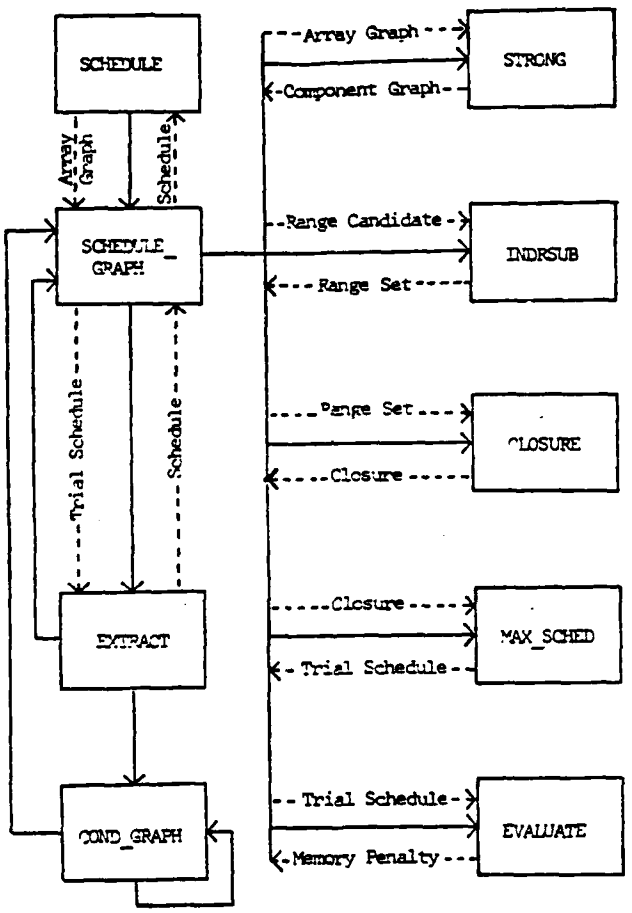


Fig. 6.9 Various components of the scheduling algorithm

<u>Global Data Structure for SCHEDULE</u>

The reduced form Array Graph, constructed by the SCHEDULE procedure, consists of a list of elements of type GNODE, with the following fields:

NXT_GNODE - A pointer to the next element in the list. (At the generation of the reduced form Array Graph all the GNODEs

form a single list. During the process separate lists will link the GNODEs in each MSCC.)

NODE_ID - The node number of the element in the dictionary.

SUXL - A pointer to a list of edges connecting this element to its successors. Initially this is identical to the SUCC_LIST list. As the process proceeds, some of the edges are removed from this list.

The components in the reduced Array Graph are found by the procedure STRONG. STRONG modifies the list connecting the nodes in the Array Graph to form separate lists for each MSCC.

The initial number of components in a Component Graph is denoted as COMP_CNT. Every component is assigned a component number from one to COMP_CNT. The component graph is defined in the following four vectors.

1) NODELST(COMP_CNT). Points to a list of GNODE elements in the Array Graph which belong to the respective component.
2) ACOMP(COMP_CNT). A boolean value showing whether the component exists in the component graph or not. In the course of the process, when a component is merged into some other component, its corresponding ACOMP bit is reset.
3) INCMP(COMP_CNT). A boolean value showing whether a component has been scheduled or not. Once a component has been scheduled, its corresponding bit will be reset. Thereby it will not be scheduled again.
4) CEDGES(COMP_CNT). Points to a list of edges which originate from the component and end at its successor components. Every element in the list has two fields. One field contains the component number of its successor and the other is a pointer which points to the next edge.

A subgraph of the Component Graph can be represented by a bit vector like INCMP. If a component is in the subgraph, its corresponding bit will be set. Otherwise, the corresponding bit will be reset. In the following, all the subgraphs of the Component Graph will use this representation.

The finally generated program schedule is structured as a list of schedule elements. There are four types of schedule elements: node-element, for-element, simul-element, and cond-element. A node-element corresponds to a primitive program event in the generated program such as the computation of an assertion, opening a file, reading a record. A for-element corresponds to a loop in the program. The body of the loop is also represented by a schedule list and pointed to from the for-element. Similarly, a simul-element corresponds to an iterative computation for a simultaneous block and points to a list in the body of the iteration. The cond-element is used to represent a conditionally executed block which corresponds to the scope of a subrange. It will point to the respective body list.

1) A node-element is a structure NELMNT, with the following fields:
   NXT_NLMN - Pointer to the next element in the schedule.
   NLMN_TYPE - Equal to 1, denoting this is a node-element.
   NODE$ - The node number.
2) A for-element is a structure FELMNT, with the following fields:
   NXT_FLMN - Pointer to the next element in the schedule.
   FLMN_TYPE - Equal to 2, denoting this is a for-element.
   ELMNT_LIST- Pointer to a program schedule which is the body of the

loop.

FOR_NAME - The dictionary node number of the loop variable.

FOR_RANGE - The dictionary node number where the range of the loop variable is specified.

3) A simul-element is a structure SELMNT which is used for a simultaneous equation block. It has the same structure as FELMNT with FLMN_TYPE equal to 3.
4) A cond-element is used for a conditionally executed block. It has a similar data structure as FELMNT except that the field FLMN_TYPE is always equal to 4.

<u>Algorithm</u> <u>6.1</u> SCHEDULE_GRAPH

Input.

G: A pointer to the reduced Array Graph which is represented by a GNODE list.

L: The nesting level L.

Output.

A program schedule for the input graph G.

Data Structures.

GSIZE(COMP_CNT): The number of nodes in a component.

MINFREE(COMP_CNT): The minimum of the number of unscheduled dimensions associated with any node in a component.

SUBRNGR($RNG_SET,$RNG_SET): A boolean matrix which shows the subrange relationships. If the jth range set is a subrange of the ith range set, then SUBRNGR(i,j) will be set to '1'B.

RNG_VEC($RNG_SET): For each range set, it indicates the node number of the indirect indexing vector which reduces the major range into this range set, if any.

1. Call procedure STRONG to find out all the MSCCs in the Array Graph G and then construct a Component Graph with each MSCC as a node. Initially all the components are put in the Component Graph and the corresponding ACOMP and INCMP bits are set to '1'B.
2. For each component, compute the corresponding element of the vector GSIZE, which is the number of nodes in the component, and the corresponding element in the vector MINFREE, which is the minimum of the number of unscheduled dimensions associated with any node in the component. Also compute the SUBRNGR matrix by scanning the indirect subscript expressions used in the assertions, and the vector RNG_VEC which gives for each range set number the node number of the indirect subscript, if any.
3. If a component has MINFREE=0, it is not to be scheduled in any loop. We will mark it off from the Component Graph by setting the corresponding INCMP bit to '0'B. This component will be a single component block.
4. Repeat step 5 to 11 to schedule all the outer level loops, until all components in the Component Graph have been marked off.
5. Select the ranges of node dimensions which are not yet scheduled and where the respective range does not have real arguments of unscheduled subscripts. The selected ranges can be scheduled in the outer level loops. The ranges of those node dimensions will be the candidate ranges.
6. Repeat step 7 to 10 for each range candidate. Steps 7 to 10 consist of a trial scheduling of a range candidate Ri.
7. Call procedure INDRSUB. This procedure computes a subgraph S which

contains all the components which are in the range set of Ri or the range set of a subrange of Ri. S is represented as a bit map similar to INCMP.

8. Call procedure CLOSURE to find the subgraph S'=closure(S).
9. Call procedure MAX_SCHED with subgraph S' and range candidate Ri as input parameters to form a loop scope Li which contains a subgraph of S'. Li is represented as a bit map similar to INCMP.
10. Call procedure EVALUATE to compute the memory penalty of Li.
11. Choose the loop Lj with the smallest memory penalty. Merge all the components in Lj into one component, say Ck, by modifying the list pointed to by the NODELST of Ck to include all the GNODEs in the other merged components. ACOMP, INCMP, and CEDGES vectors are also modified to reflect the new component. Then set INCMP(k) to '0'B to mark the whole loop scope off from the Component Graph.
12. Do a topological sort over the resulting components of the component graph where each component corresponds to either a single node or a loop scope in the schedule to be returned.
13. Schedule each component separately. If there is no distinguished dimension for the nodes in a merged component, a node-element will be formed for the component. Otherwise, call the procedure EXTRACT to form a for-element for the component.

<u>Algorithm</u> <u>6.2</u> STRONG
Input.
    G: A pointer to an Array Graph.
Output.
    NODELST: A list of components which are the MSCCs of the input graph. Every component is represented by a list of GNODE elements which belong to the component.

```
 1. Clear the stack, the component count, the list of components
    NODELST, and the variable COUNT. For each node v in the graph G set
     DFNUMBER(v) = 0
2. For each node v in the graph G such that DFNUMBER(v)=0 call SEARCH(v)
   to add the components reachable from v to the component list NODELST.
3. Return the component list as the result.
```

<u>Algorithm</u> <u>6.3</u> SEARCH
Input.
    v: A node in a graph which is not examined yet.
Output.
    The NODELST for all the MSCCs reachable from node v.

```
 1. Set COUNT to COUNT+1 and DFNUMBER(v), LOWLINK(v) to COUNT. Push v
    on the stack.
 2. Repeat the following substeps for each node w, a direct descendant
    of v.
    2.1 If DFNUMBER(w)=0, call SEARCH(w) and then let
        LOWLINK(v)=min(LOWLINK(v),LOWLINK(w)).
    2.2. Else, if DFNUMBER(w)>0 and w is on the stack, then let
        LOWLINK(v)=min(DFNUMBER(w),LOWLINK(v)).
 3. If LOWLINK(v)<DFNUMBER(v) then return.
 4. Else, LOWLINK(v)=DFNUMBER(v). Node v is a root of a strongly
    connected component. All the elements (above and including v) on
    the stack are successively popped off the stack and linked into a
    list - a subgraph which is defined as a component. This component
    is placed on the top of a list of components pointed to by the
```

variable COMP_LIST. In addition a unique component number is assigned to each node w in the current component.

<u>Algorithm</u> <u>6.4</u> INDRSUB(RANGE,GI)

Input.

RANGE: A candidate range (a range set number).

Output.

GI: A subgraph which contains all the components in the range set of RANGE and the components in the range sets of the subranges of RANGE which can be included in the loop scope of RANGE.

1. Construct a subgraph GI which contains all the components in the Component Graph which have an unscheduled dimension with the range RANGE. GI is represented in a bit vector similar to INCMP. Set GI(k)='1'B if the kth component is in the range set of RANGE. The edges from these nodes are given in CEDGES.
2. If RANGE has no subranges, return GI as the result. This information stored previously in SUBRNGR matrix, which shows the subrange relationships.
3. Otherwise, repeat step 5 to 8 for each immediate subrange RNGIK of RANGE.
4. Call INDRSUB recursively with RNGIK as input parameter and GIK as the output parameter. GIK will contain the components which can be scheduled in the loop of RNGIK.
5. Call procedure CLOSURE to compute the closure of GIK in the Component Graph. Then put the closure into GIK.
6. Set the union of GI and GIK into GI. (Note that this may be reversed in step 8.)
7. Call MAX_SCHED procedure to do a trial scheduling for subgraph GI.
8. If the subgrpah GI can not be scheduled completely, then at least one node, and possibly more, will have to be physical. Also the range specification of the subrange may become necessary. Therefore we decided that in this case it is not worthwhile to merge the range set of RNGIK with the range set of RANGE and GIK is taken out of GI.
9. Return GI as the result.

<u>Algorithm</u> <u>6.5</u> CLOSURE(COMPS)

Input.

COMPS(COMP_CNT): A bit vector with a set of components marked by '1'B. Other components are marked by '0'B.
The algorithm also uses the global data structures (ACOMP and CEDGES).

Output.

CCOMPS: A bit vector with the closure of the set of components in the input marked by '1'B. Other components are marked by '0'B.

1. Create a bit vector NACOMP (size COMP_CNT) with the components in ACOMP marked except the components in COMPS are merged into one component. This also involves creating a vector NCEDGES similar to CEDGES except reflecting the merger of the components in COMPS.
2. Find all the MSCCs in the new component graph (consisting of the new vectors NACOMP and NCEDGES).
3. Locate the MSCC which includes the components in COMPS.
4. Construct CCOMPS, a bit vector (size COMP_CNT), with all the components in the MSCC marked. This is the closure set of the input.

Algorithm 6.6 MAX_SCHED

Input.

INCMP: A bit vector where a set of yet unscheduled components is marked by '1'B. Other scheduled components have a value '0'B. Note that these unscheduled components are the basic MSCCs found by STRONG. The function of MAX_SCHED is to schedule as many of the marked components as possible.

MERGCMP: A bit vector with the closure of a range set marked by '1'B.

RANGE: The candidate range (range set number).

Output.

COMPS: A bit vector with the components, which have been trial scheduled in a loop, marked by '1'B.

POSITION: A vector (size is DICTIND- the number of nodes in the dictionary). The position in each scheduled node of the distinguished dimensions that corresponds to the loop parameter.

1. Initialize the POSITION entries to 0.
2. For each component i, if INCMP(i)='1'B (i.e. it is not yet scheduled), MERGCMP(i)='1'B (i.e. it is in the closure set), then search the CEDGES vector and set PREDCNT(i) to number of predecessors in MERGCMP. If PREDCNT(i)=0 then put component i into a list of candidates to be trial scheduled.
3. Repeat steps 4 to 8 until the list (referred to in step 2) is empty. The function of steps 4 to 8 is to merge one component from the list into the loop scope represented by COMPS.
4. Remove a component, say Ci, from the list. Search through the NODELST of Ci, if there exists a node v with POSITION(v)>0 (i.e. its distinguished dimension has been determined in a previous iteration), then set FIRSTNODE=v, and go to step 7.
5. Else, arbitrarily pick any node of the component. Let it be denoted by v. Set FIRSTNODE=v.
6. Search the subscript list of node v until finding a dimension j that has not been scheduled in a loop scope (i.e. IDWITH=0) and its range is the same as the RANGE parameter. If found, then POSITION(v)=j. If none found then this component should not be scheduled in the loop scope. Therefore go to next iteration (i.e. end of step 9).
7. Propagate the distinguished dimension of node v repeatly until all the nodes in Ci have their distinguished dimensions defined. During each propagation step:
   7.1 Propagate the distinguished dimension forward along the edges originated from node v to all the nodes at the terminating end of the edges.
   7.2 If the node to which a distinguished dimension is propagated does not belong to Ci then do not further propagating the distinguished dimension from this node forwards.
   7.3 If propagation is not possible to any node in Ci because of type 4 subscript expression then the current iteration may be terminated, i.e. go to end of step 9.
8. The current component can be merged into the loop scope. Set COMPS(i)='1'B.
9. Search through the list pointed by CEDGES(i). For every edge from Ci to Ck set PREDCNT(k)=PREDCNT(k)-1. If PREDCNT(k)=0, INCMP(k)='1'B, and MERGCMP(k)='1'B, then put Ck into candidate list.

Algorithm 6.7 EVALUATE

Function: Given a loop scope, compute the resulting penalty in use of memory. This procedure is called after each trial schedule for a range candidate and again after the final schedule was selected.

Input.

COMPS: A bit vector of size COMP_CNT with the bits correspondning to components in a loop scope equal to '1'B.

EVAL_SET: A bit denoting whether EVALUATE is called to evaluate memory penalty of a trial schedule or for the selected schedule, in which case the selected memory allocations are recorded in STOTYP.

Output.

PENALTY: The memory penalty of the loop scope, in bytes.

Data structure.

SRCPHY, TGTPHY: When an edge in an Array Graph crosses a boundary of a loop scope then, depending on the type of the edge, the memory allocation for the data node at the origin or terminating ends of the edge may have to be physical. The SRCPHY bit vector denotes for each type of edge (there are 28 types) whether the memory allocated to the node at the origin end of the edge (the source node) must be physical. Similarly, the TGTPHY vector refers to the node at the terminating end of the edge (the target node).

MRAL: The memory requirement, in bytes, after the loop is formed.

MRIC: The memory requirement in the ideal case.

STOTYP: A field in the data structure LOCAL_SUB. For a virtual dimension, STOTYP=0. For a window of width k+1 dimension, STOTYP=k+1. For a physical dimension with upper bound u, STOTYP=-u.

1. Repeat steps 2 to 6 for every edge in the Array Graph. Each iteration computes the effect of the edge on use of memory.
2. If the source and the target nodes of the edge are in COMPS, this is an internal edge, then go to step 6 to examine the subscript expression of the edge to determine its effect on use of memory.
3. If both the source and the target nodes of the edge are not in COMPS, then this edge has no effect on memory useage. Go to end of iteration, at end of step 6.
4. If none of the above then this edge crosses the loop boundary. In this case, if SRCPHY(EDGE_TYPE)=1, then the distinguished dimension of the source node must be physical. If TGTPHY(EDGE_TYPE)=1, then the distinguished dimension of the target node must be physical. The respective node numbers and the requirements for physical memory allocation are stored in a list. Also in this case go to the end of the iteration (at end of step 5).
5. If the subscript expression is of the form I-k and SRCPHY(EDGE_TYPE)=1, then the memory allocation for the distinguished dimension of the source node must be a window of width k+1. This is also stored in the list.
6. PENALTY is initialized to zero.
7. Repeat steps 8 to 11 for every node in the above list. These nodes have either a physical or window of width k+1 memory allocation. An iteration computes the memory requirement for a respective node.
8. In the case of a physical distinguished dimension, compute MRAL, as the product of all the ranges of the unscheduled node subscripts.

In the case of a window of width k+1 for the distinguished dimension, compute MRAL as the product of k+1 and the ranges of the other unscheduled node subscripts.

9. To compute MRIC it is necessary to scan each unscheduled node subscript. If its storage type STOTYP is 0, then the ideal memory requirement for this dimension is one. If STOTYP<0, the memory allocation has previously been determined as physical, then the ideal memory requirement is -STOTYP (u). MRIC is the product of these ideal ranges.
10. The penalty for the array node ND_PENALTY= (MRAL-MRIC)*(length of node element in bytes).
11. PENALTY=PENALTY+ND_PENALTY.
12. If EVAL_SET='1'B then if the distinguished dimension is physical then STOTYP in every unscheduled dimension is equal to the minus of its range, if the distinguished dimension is a window of width k+1 then STOTYP of the distinguished dimension is k+1 and for the other unscheduled dimensions STOTYP is the minus of their respective range.

<u>Algorithm</u> <u>6.8</u> EXTRACT

Function: To obtain the for-element for a loop, including the schedule elements for the body of the loop scope.

Input.

SUBGRAPH: A pointer to a reduced Array Graph of the component scheduled into one loop scope.

SVPOSITION: A vector with an element for every node in the SUBGRAPH. Each element has the value of the dimension number of the distinguished dimension of the respective node.

L : The nesting level.

Output.

A for-element which is the schedule of the input graph.

1. Allocate a for-element. Set FOR_NAME to loop parameter name and FOR_RANGE to the range set number of the loop parameter.
2. If the current loop range has some immediate subranges, then call procedure COND_GRAPH and upon return go to step 7. COND_GRAPH takes over all further scheduling of a body of a loop which contains conditionally executable nodes due to use of indirect subscripting.
3. Delete all the edges from the graph with distinguished dimension subscript expressions of type 2 or 3. The precedence expressed by these edges is assured by the order of the iterations.
4. Set IDWITH of the distinguished dimension of all the nodes in the subgraph to L, the nesting level of the current loop.
5. Call SCHEDULE_GRAPH, with SUBGRAPH and L+1 as the parameters, to get the schedule of the resulting graph.
6. Set ELMNT_LIST in the for-element structure to point to the schedule returned from step 5.
7. Return the for-element as output.

<u>Algorithm</u> <u>6.9</u> COND_GRAPH(TOP_RANGE,GRAPH)

Function : To obtain the schedule elements of the body of a loop scope, which includes cond-elements.

Input.

TOP_RANGE: The range set number of the highest level major range in the SGRAPH.

SGRAPH: A graph to be scheduled within an iteration block of the range TOP_RANGE.

Output. A schedule for SGRAPH.

1. Scan all edges in SGRAPH. If an edge has a subscript expression in the distinguished dimension of types 2, 3, 6, or 7, and either the source or the target nodes have the TOP_RANGE range, then delete this edge from SGRAPH.
2. If node X is the indirect indexing vector served to reduce the range TOP_RANGE to a subrange RNGIK, then draw an edge from X to all the nodes in the range set of RNGIK.
3. Call procedure STRONG to form a Component Graph for SGRAPH, consisting of ACOMP and INCMP, CEDGES, and NODELST. ACOMP and INCMP are bit vectors (the size is the number of MSCC found by STRONG). These vectors are all of the value '1'B.
4. For every subrange RNGIK of TOP_RANGE, merge all the components in the range sets of RNGIK or its direct and indirect subranges into one component. Set the INCMP vector elements of the merged components to '0'B.
5. Repeat steps 6 to 9 until all the elements in INCMP are '0'B. Each iteration merges a group of components with TOP_RANGE range.
6. Call CLOSURE with INCMP to obtain the closure set MERGE_CMP.
7. CALL MAX_SCHED with INCMP, MERGE_CMP, and TOP_RANGE. It returns CCOMPS.
8. Merge the components in CCOMPS into one component, updating NODELST, CEDGES, ACOMP, and INCMP.
9. Set the element of INCMP corresponding to the merged schedule to '0'B.
10. Repeat steps 12 to 13 for the components in ACOMP.
11. Select the next component in ACOMP in a topologically sorted order. Let this component be COMPI.
12. Let RNGIK be the range of the component COMPI. If RNGIK=TOP_RANGE, then mark the distinguished dimension of each node in the component as scheduled and call procedure SCHEDULE_GRAPH to get a schedule for this component. Go to step 14.
13. Otherwise, allocate a cond-element to this component. Call procedure COND_GRAPH recursively with RNGIK and COMPI as the input parameters to get a schedule for the conditional element.
14. Return the schedule elements obtained as the final schedule of SGRAPH. Note that the order of the schedule elements was determined by the selection of components in a topologically sorted order in step 11. The schedule elements are obtained either in step 12 or 13, depending on whether they are cond-elements or other elements respectively.

CHAPTER 7

CODE GENERATION

7.1 OVERVIEW OF THE CODE GENERATION PROCESS

Code Generation is the last phase of the processor. It uses the data structure generated in Array Graph construction, specification analysis, and program scheduling. As shown in Fig. 7.1 the code generation process accepts two inputs: the program schedule created in the scheduling phase and attribute tables produced in the analysis phase. Recall that the program schedule is an ordered sequence of schedule elements described in section 6.6. The nodes referenced in schedule elements can be found in the dictionary. The attributes of the respective nodes are in the dictionary. They are described in the section 4.2.1. The output is a complete PL/I program ready for compilation. The executable PL/I code is written out to the "PL1EX" file. The PL/I "ON" conditions are written to the "PL1ON" file and the PL/I code for declaring the object data items is written to a "PL1DCL" file.

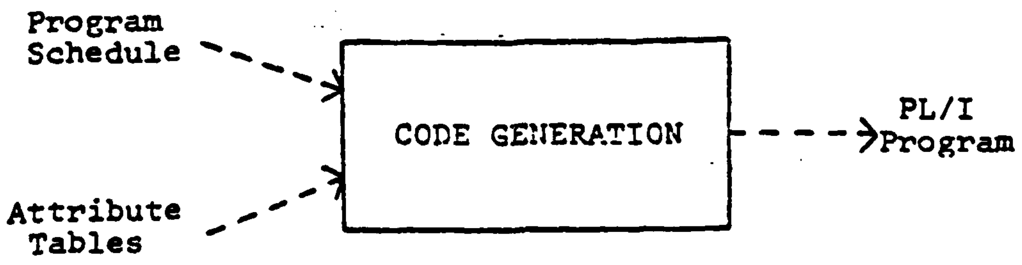


Fig. 7.1 Overview of the Code Generation Phase

Fig. 7.2 shows the overall organization of the code generation process, consisting of the main procedure CODEGEN which in turn calls on the other procedures to perform certain tasks. The PL/I execution code is generated by the GENERATE procedure which examines the elements of the schedule one at a time, and invokes the procedures that are indicated by types of program events. The GPL1DCL procedure generates the data declarations. GENERATE calls GEN_NODE to generate statement for node elements of the schedule. The GEN_NODE calls on GENIOCD for input-output operations and on GENASSR for assertions. GENERATE also calls GENDO and GENEND for generating iteration control structures for for-elements, and on COND_BLK and COND_END for generating conditional block statements for cond-elements. These procedures are briefly reviewed in section 7.2. They are described in greater detail together with other auxiliary tasks in the subsequent sections that follow.

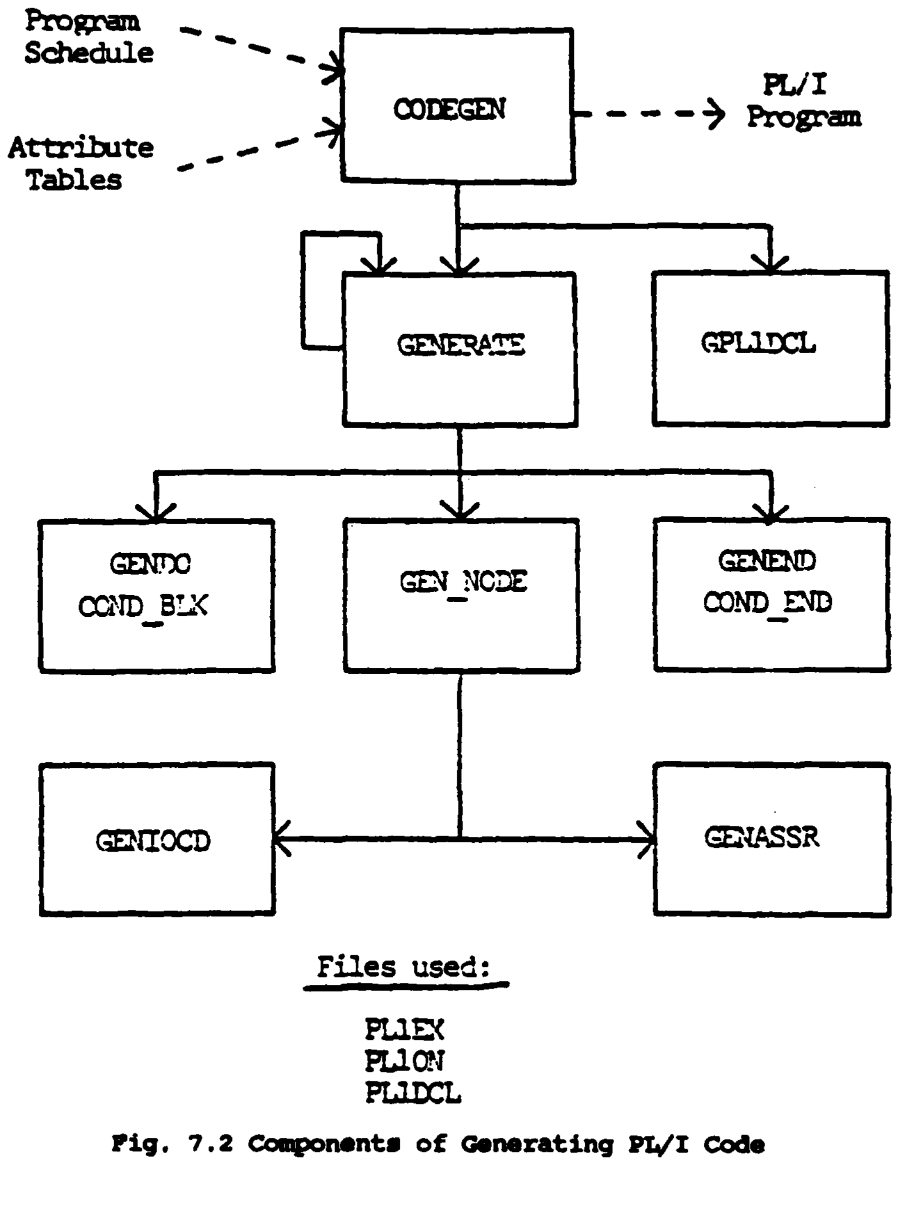


Fig. 7.2 Components of Generating PL/I Code

7.2 THE MAJOR PROCEDURES FOR CODE GENERATION

7.2.1 CODEGEN - THE MAIN PROCEDURE

CODEGEN starts with opening the output files PL1EX, PL1ON, and PL1DCL. It next generates code that will handle program errors. Most of these errors are due to input data errors discovered by data type conversions in the program. The user can also define additional error conditions. The statements written to the PL1EX file are as follows:

```
        ALLOCATE ERROR, ACC_ERROR ;
        ACC_ERROR = '0'B ;
        ALLOCATE $ERR_LAB ;
        $ERR_LAB = END_PROGRAM ;
```

The declarations written to the PL1DCL file are as follows:

```
        DCL (ERROR, ACC_ERR, NOT_DONE) CTL BIT(1) ;
        DCL $ERR_LAB LABEL CTL ;
```

Finally the ON condition code is sent to the PL1ON file as follows:

```
        ON ERROR
        BEGIN
          /* write erronous input record to ERRORF file */
          WRITE FILE(ERRORF) FROM($ERROR_BUF) ;
          ERROR = '1'B ;     /* set error flag           */
          GO TO $ERR_LAB ;   /* go to end of loop where */
        END ;                /* error was detected       */
        ERROR_RESTART:
```

CODEGEN next passes the entire program schedule to GENERATE, which will generate the portions of the program for the schedule elements. When this is completed CODEGEN passes the attribute tables to GPL1DCL to generate data declarations. Finally CODEGEN calls on MERGEPL1 to merge the three output files.

7.2.2 GENERATE - INTERPRETING SCHEDULE ELEMENTS

This recursive procedure scans the schedule given by the list of schedule elements, LIST, for a loop nesting level LEVEL. To start with, CODEGEN passes the whole schedule at level 0. In subsequent calls GENERATE will receive a schedule of a loop scope at each nesting level. GENERATE calls lower level procedures to process the different types of schedule elements as follows:

1. Scan each element of the list LIST. For each element perform steps 2 to 4.
2. If the element is a node-element call GEN_NODE which will generate the code for the schedule element.
3. If the element is a for-element do the following:
   3.1 Call GENDO to produce a code for opening a loop.
   3.2 Call GENERATE recursively with the list of the elements within the loop's scope and level = LEVEL+1.
   3.3 Call GENEND to generate the termination of the loop.
4. If the element is a cond-element do the following:
   4.1 Call COND_BLK to produce the code for opening a conditional block.

4.2 Call GENERATE recursively with the list of the elements within the condition block and level = LEVEL.
4.3 Call COND_END to generate the termination of the conditional block.

7.2.3 GENDO - TO INITIATE THE SCOPE OF ITERATIONS

This procedure produces the code for a control statement initiating an iteration loop. The loop variable name FORNAME and the termination criterion are taken from the fields FOR_NAME and FOR_RANGE in the for-element being scanned.

The following instructions are intended for recovery from a program error. They always precede each loop control statement:

```
        ALLOCATE ERROR, ACC_ERROR ;
        /* reset accumulative error flag */
        ACC_ERROR = '0'B ;
        ALLOCATE $ERR_LAB ;
        $ERR_LAB = LOOP_ENDc ;
```

The "c" following LOOP_END is a unique number assigned to the loop. The purpose of these statements is to ensure that an error occurring within the loop scope will cause the control be directed to LOOP_ENDc which is a label immediately preceeding the end of the loop.

The DO-statement itself is constructed next. Two basic forms for the loop control statements are used:

1)
```
    DO name = 1 TO upper [ WHILE (condition) ] ;
```
2)
```
    name = 0 ;
    DO WHILE (condition) ;
      name = name+1 ;
```
"name" is the loop variable. "condition" is the termination condition.

If the termination criterion given is that of a fixed upper limit or given through a SIZE variable, the first form is used and "upper" is either a constant number or a variable of the form SIZE$X.

If the range is specified by an END.X control variable, the second form of loop control is used. In this case we use NOT_DONE in the condition and the following statements are generated before the beginning of the loop:

```
        ALLOCATE NOT_DONE ;
        NOT_DONE = '1'B ;
```

NOT_DONE will be reset to '0'B whenever the appropriate END.X variable is set to 'true'.

If there is an end-of-file condition associated with the iteration, either as the main termination condition, or because this is an iteration on an input record or group above the record level which are last in their peer group, we add:

```
        ^ENDFILE$file
```

to the condition "condition".

7.2.4 GENEND - TO TERMINATE THE SCOPE OF ITERATIONS

This procedure produces the code needed at the end of the loop scope. Since at times, we use k+1 locations to store a window of size k+1 of an array, it is necessary on each iteration to shift the window by one element position. This is done at the end of the iteration. The size of respective window is originally stored in STOTYP of the node subscript of each array node. GENERATE passes the node numbers of arrays using window dimensions in a list called PREDLIST to GEN_END. Based on this list GEN_END generates statements to shift the window by one element position. The actual range declared for a window dimension is k+1. In each iteration we compute (or read) A(..., k+1, ...) and may refer to the previous element as A(..., k, ...). When an iteration is completed we transfer A(..., I+1,...) to A(..., I,...) for I from 1 to k.

After producing a sequence of these shifting operations we produce the label:

```
        LOOP_ENDc: ;
```

where "c" is the unique count associated with the current loop. If the termination criterion for the loop was through an END.X control variable we also produce the code:

```
        IF END.X THEN NOT_DONE = '0'B ;
```

This has to be done at the end of the loop since the value of END.X at a given iteration determines whether this iteration will be the last.

After this we produce the following statements:

```
        $TMP_ERROR = ACC_ERROR ;
        FREE ERROR, ACC_ERROR ;
        FREE $ERR_LAB ;
        IF $TMP_ERROR THEN ERROR, ACC_ERROR = '1'B ;
```

If the termination criterion was through an END.X control variable we also produce:

```
        FREE NOT_DONE ;
```

7.2.5 COND_BLK - INITIATE A CONDITIONAL BLOCK

This procedure produces the code necessary to initiate a conditional block. The conditional block will be executed within the iteration only when the value of the indirect subscript is increased. The indirect subscript node number is stored in the FOR_RANGE field of the cond-element being scanned. An IF-statement is generated to test the above condition. Inside the conditional block we will use a new symbol for the indirect subscript. For example, if X(I) is the indirect subscript then we define a new subscript J=X(I). Let 'old-sub' denote the subscript running in the major range, i.e. I. The 'new-sub' denotes the new representation of the indirect subscript, i.e. J. A boolean variable, $B_X, indicates whether the conditional block should be executed. The code to compute $B_X is generated by GEN_NODE when the node X is scanned in the schedule. The new-sub is of the form $Xn where 'n' is a unique number associated with this conditional block. The following declaration statements are issued:

```
          DCL $Xn FIXED BIN ;
          DCL $B_X BIT(1) ;
```

The following codes is then produced:

```
          IF $B_X THEN DO ;
            new-sub = X( ..., old-sub) ;
```

7.2.6 COND_END - TERMINATE A CONDITIONAL BLOCK

This procedure produces the code at the end of a conditional block. The above IF-statement has been generated by COND_BLK. Here we issue an 'END' statement to terminate the IF-statement.

7.3 GEN_NODE - CODE GENERATION FOR A NODE

This procedure generates the code associated with a schedule node-element. It branches to different parts according to the types of nodes.

7.3.1 PROGRAM HEADING

If the node is a module name (type MODL) we produce the code:

```
          name: PROCEDURE OPTIONS(MAIN) ;
```

This code is routed to the file PL1DCL.

7.3.2 FILES

If the node is a file node (type FILE) we first generate three names. "file_stem" is the file name with prefixes "NEW" or "OLD" removed, if any. "name" is the full name of the node, including all prefixes. "file_suff" is the file_stem with the suffix of 'S' for source file, 'T' for target file, and 'U' for update file (both source and target). The following declaration statements are routed to PL1DCL file.

```
          DCL name_S CHAR(length) VARYING INIT(' ') ;
          DCL name_INDX FIXED BIN ;
```

"length" is the maxmimum length of records in the file. "name_S" is the name of a buffer into which records in the file are read. (It is

VARYING as the file may have more than one record type, with different lengths.) "name_INDX" is a variable used to scan the buffer for packing and unpacking the records (explained further later).

1. If the file is an input file we produce the statement:
   ```
   OPEN FILE (file_suff) ;
   ```
2. If the file is a sequential input file and an end-of-file is not explicitly mentioned by the user, we produce the declarations:
   ```
   DCL ENDFILE$file_stem BIT(1) INIT('0'B) ;
   DCL $FSTfile_suff BIT(1) INIT('1'B) ;
   ```
   routed to PL1DCL file. If the user explicitly mentioned the end-of-file variable then these statements will be generated when the declaration are generated for all variables by GPL1DCL.

   The statements:
   ```
   ON ENDFILE (file_suff)
   BEGIN
     ENDFILE$file_stem = '1'B ;
     name_S = COPY(' ',length) ;
   END ;
   ```
   are sent to PL1ON file. The purpose of these statements is to have the file buffer filled with blank characters when an end of file condition occurs.
3. If the file is an output file we produce the statement:
   ```
   CLOSE FILE(file_suff) ;
   ```

7.3.3 RECORDS

If the node is a record (type RECD) we call GENIOCD to produce the code for the reading or writing of records.

7.3.4 FIELDS

To process fields GEN_NODE calls procedure GENITEM. GEN_NODE also calls CHECK_VIRT to find if the node has a windowed dimension. If the field node is an indirect subscript, X, the following code is issued.

```
IF loop_var=1 THEN DO ;
  bname = '1'B; rname = 0; END ;
ELSE IF X(loop_var)>X(loop_var-1) THEN DO ;
  bname = '1'B; rname = 0; END ;
ELSE DO ;
  bname = '0'B; rname = 1; END ;
```

where loop_var is the current level loop variable, bname is of the form $B_X, and rname is of the form $R_X. Recall that bname indicates whether the associated conditional block will be executed. rname will be used to compute the index to reference an element such as A(X(loop-var)) in the case that array A has a windowed dimension. This is explained further later in connection with the code generation for assertions.

7.3.5 ASSERTIONS

If the node is an assertion we call the procedure GENASSR to produce the code for an assertion.

7.4 GENASSR - GENERATING CODE FOR ASSERTIONS

This procedure generates code for assertions. The main task of GENASSR is to transform the syntax tree representation of the assertion into a string representation acceptable by the PL/I compiler. The transformation is carried out by a recursive climb on the syntax tree, combining for each node the string representations of the descendant subtrees into a string representation of the tree rooted at that node. However, before performing the main task the procedure transforms assertions containing conditional expressions into conditional assertions. Thus, an assertion of the form:

```
          Y = IF (IF X>0 THEN Y>0 ELSE Y<=0) THEN X*Y
                                             ELSE -X*Y ;
```

will be transformed into:

```
          IF X>0 THEN IF Y>0  THEN Y = X*Y ;
                              ELSE Y = -X*Y ;
                 ELSE IF Y<=0 THEN Y = X*Y ;
                              ELSE Y = -X*Y ;
```

The overall execution of GENASSR can therefore be summarily described as:

1. Transform assertions with conditional expressions into conditional assertions.
2. Form the string representation of the assertion.

7.4.1 TRANSFORMING CONDITIONAL EXPRESSIONS

This task is carried out by the procedure SCAN which uses the auxiliary procedure EXTRACT_COND.

7.4.1.1 SCAN (IN)

The procedure SCAN effects the complete transformation of assertions containing conditional expressions into conditional assertions. The procedure is presented with an assertion pointed to by IN, and returns a pointer to the transformed assertion. The steps in this procedure are as follows:

1. Check the root of the tree pointed to by IN to see whether it is a simple assertion or a conditional assertion. If it is a simple assertion then go to step 5.
2. We check next if the conditional assertion contains conditional

expressions. A conditional assertion has the form:
IF COND THEN S1 ELSE S2
where S1, S2 are assertions.
SCAN calls EXTRACT_COND to check whether *COND contains* a conditional expression. If COND contains a conditional expression, then EXTRACT_COND returns C, L, and R which are the parts of COND as follows:

```
        COND = IF C THEN L ELSE R.
```

Otherwise, go to step 4.

3. If a conditional expression is found in COND then:
   3.1 SCAN then transforms the tree (pointed to by IN) into a tree IN1 which consists of the form:
   ```
            IF C THEN IF L THEN S1
                           ELSE S2
                 ELSE IF R THEN S1
                           ELSE S2
   ```
   3.2 SCAN calls SCAN(IN1) recursively to further search for conditional expressions in *IN1* and return a transformed conditional assertion.
   3.3 The transformed assertion is returned by SCAN.
4. If COND does not contain embedded conditional expressions, then there are two recursive calls to SCAN for the assertions S1 and S2 in IN. SCAN then returns the following assertion and exits.
   ```
        IF COND THEN SCAN(S1) ELSE SCAN(S2)
   ```
5. In the case of a simple assertion:
   ```
        Y = E.
   ```
   SCAN calls EXTRACT_COND(E) to search for conditional expressions in E. If none found, then assertion Y = E is returned unchanged. Otherwise, EXTRACT_COND returns C, L, and R which are the parts of E as follows:
   ```
        E = IF C THEN L ELSE R.
   ```
6. If E contains conditional expression, then SCAN calls SCAN(IN2) recursively, where IN2 points to a tree of an expression of the form:
   ```
        'IF C THEN Y = L
              ELSE Y = R'
   ```
   The return from the recursive call on SCAN is returned by SCAN as the transformed assertion.

7.4.1.2 EXTRACT_COND(ROOT,COND,LEFT,RIGHT)

This procedure identifies and extracts the leftmost conditional expression in a given expression pointed to by ROOT.

If a conditional expression is found the (pointer to the) condition is returned in COND and its first (THEN) and second (ELSE) subexpressions returned in LEFT and *RIGHT* respectively. If the analyzed expression contains no conditional expression the procedure returns NULL in COND.

Its operation is as follows:

1. Inspect the top level node of the given syntax tree.
2. If it is a conditional expression, return respectively the condition,

the subexpression following THEN, and the subexpression following ELSE, then exit.

3. If the expression is a simple expression, i.e. a constant or a variable, return NULL and exit.
4. If the expression is a compound expression, scan each of its descendants by calling EXTRACT_COND recursively. Consider the first COND, LEFT, and RIGHT which are returned such that COND is not equal to NULL. In general, a compound expression is of the form:

   E = g(E1,...,Em)

   Assume that the recursive scanning of E1, ..., Em produces first COND not equal to NULL for Ei where 1<=i<=m, returning also the THEN and ELSE subexpressions L, and R respectively. Then the current call for E returns:

   COND as the condition,
   g(E1, ...,Ei-1,L, ...,Em) as LEFT, and
   g(E1, ...,Ei-1,R, ...,Em) as RIGHT.

Thus the overall effect of EXTRACT_COND on an expression E is to extract a condition C if one exists in E (returned as COND), and then to compute E1 when C is true, and E2 when C is false. E1 and E2 are returned in LEFT and RIGHT respectively. Described in another way we look for C, E1, and E2 such that the following equivalence holds:

```
        E = IF C THEN E1 ELSE E2 .
```

In particular this gives:

```
        g(E1, ...,Ei-1,(IF C THEN L ELSE R),...Em) =
               IF C THEN g(E1, ...,Ei-1,L,...,Em)
                    ELSE g(E1, ...,Ei-1,R,...,Em).
```

7.4.2 PRINT - TRANSFORMING THE ASSERTION INTO STRING FORM

This procedure is presented with a pointer to an assertion syntax tree and it converts the assertion tree into a string representation.

The procedure branches according to the types of the nodes in the assertion tree.

1. If the node is a subscripted variable A(E1,...,Em) we generate the string 'A('. We then scan each of the subscript expression E1 to Em and add them to the string according to the following subcases:
   1.1 If the dimension at position i corresponds to the dimension declared for repetition of a record and the variable A includes the prefixed 'NEXT', then
      1.1.1 If the dimension is scheduled as a window of width k+1 we insert the subscript value k+2.
      1.1.2 If the dimension is scheduled as physical and the expression Ei is a constant c, then insert the value of c+1. (See further below.)
      1.1.3 If the dimension is scheduled as physical and Ei is an expression we call PRINT(Ei) and insert the returned value concatenated with '+1'.
   1.2 If the dimension at position i is scheduled as a window of width k+1, in this case the physical allocation for the array dimension is k+2 elements with the k+1th element standing for the current value and the k+2th element standing for the field in the next

record. The different subscript expressions are handled as follows:

1.2.1 If it is a simple subscript then we insert an integer k+1 as the subscript.
1.2.2 If the subscript expression is I-c, then an integer k+1-c is inserted.
1.2.3 If the subscript expression is X(I), then k+1-$R_X is inserted where k+1-$R_X points to the element A(X(I)). If X(I)=X(I-1) then $R_X is equal to 1, and if X(I)>X(I-1) then $R_X is equal to 0. (The code to compute $R_X is generated by GEN_NODE right after node X is scanned.)
1.2.4 If the subscript expression is X(I)-c, then k+1-$R_X-c is inserted as subscript.
1.2.5 If the subscript expression is X(I-a), then k-[X(I-1)-X(I-a)] is inserted as the subscript. X(I-1)-X(I-a) is the offset of A(X(I-a)) to A(X(I-1)) which is stored in the kth element of the window for the ith dimension of array A.
1.2.6 If the subscript expression is X(I-a)-c, then k-[X(I-1)-X(I-a)]-c is inserted as the subscript.

1.3 If the ith dimension of array A is physical and Ei is the subscript expression, we call PRINT(Ei) and insert the returned value.

2. For all other compound nodes we call PRINT recursively to convert the descendants and insert between them the string representation of the separators, operators, and delimiters. The latters are stored in the OP_CODE fields as integer codes. The integer codes are translated into the operator representation using the array KEYS and then inserted.
3. For atomic nodes we use the variable name either directly or through its node number. Loop variables (subscripts) are accessed through the level indication available in their IDWITH field which is used as an index to the array LOOP_VARS. Function names are retrieved by their function number indexing the table FCNAMES.

7.5 GENIOCD - GENERATING INPUT/OUTPUT CODE

GENIOCD is invoked by CODEGEN upon scanning a schedule element which corresponds to a record node. It accepts as input the node number in the schedule element. GENIOCD generates PL/I READ, WRITE, or REWRITE statements with the appropriate parameters, based on the attributes of the file, as well as the control code or condition code associated with the input/output operation.

Table 7.1 summarizes the different statements generated by GENIOCD for the different cases. Each of the different cases in Table 7.1 shows the conditions defining the case and the statements which are generated for the case. The upper case letters represent the part of the actual PL/I string being generated, whereas the lower case letters are the metanames of the items obtained from the program schedule elements.

Several preparatory steps are taken before branching to the different cases.

1. Definition of names: We generate several variable names derived from the record name that will be used in the code. Let the record name be designated by rec.
   1.1 If rec is of the form OLD.X or NEW.X we define recname as OLD_X or NEW_X respectively.
   1.2 Otherwise we define recname as rec.
   1.3 Recbuf is defined as recname_S.
   1.4 Recindx is defined as recname_INDX.

   Consider now the file which is parent to rec. Let it be denoted by fil.
   1.5 Set file_name to fil.
   1.6 If fil is of the form OLD.X or NEW.X set file_name to OLD_X or NEW_X respectively and file_suff to file_nameU.
   1.7 Otherwise set file_suff to file_nameS if the file is a source and to file_nameT if the file is a target.
   1.8 Set eof to ENDFILE$file_name.
   1.9 Retrieve the keyname associated with the record, if one exists, and assign it to key_name.
   1.10 Set found to FOUND$file_name.
2. Issue the following declarations.
```
        DCL recbuf CHAR (len_dat(n)) ;
        DCL recindx FIXED BIN INIT(1) ;
```
   This declares a buffer for the record into which and out of which the information will be read or written. 'Len_dat(n)' here gives the buffer length.
3. If the record is an output record, the instruction for moving the data from each field into the record buffer will be generated.
4. If the record is an output record and a SUBSET condition was specified for it we enclose the code for writing the record by the condition:
```
        IF SUBSET$rec THEN DO ;
              code
        END ;
```

The procedure DO_REC produces the code for reading and writing of records. It branches according to the cases in Table 7.1.

Table 7.1 The Various cases of program I/O control

Case 1: An Input Sequential and Nonkeyed Record.

The following code is produced:

```
      IF $FSTfile_suff THEN DO ;
        READ FILE (file_suff) INTO (recbuf) ;
        $FSTfile_suff = '0'B ;
      END ;
      ELSE recbuf = filebuf ;
      recindx = 1 ;
      IF ^ENDFILE$file_name THEN
        READ FILE (file_suff) INTO (filebuf) ;
      $ERROR_BUF = recbuf ;
```

The movement of the data to the individual fields will be done in conjunction with the nodes corresponding to the fields (see GENITEM). The next record is always read into file buffer so that we can unpack the data for the NEXT record.

Case 2: Input, Sequential and Keyed Record.

Ensure that the following reclarations have been issued:

```
      DCL FOUND$rec BIT(1) ;
      DCL PASSED$rec BIT(1) ;
```

Issue now the code:

```
      FOUND$rec, PASSED$rec = '0'B ;
      DO WHILE(^ENDFILE$file_name & ^PASSED$rec) ;
        READ FILE (file_suff) INTO (recbuf) ;
        (code for extracting the key field)
        IF keyname = POINTER$rec THEN
          FOUND$rec, PASSED$rec = '1'B ;
        ELSE IF keyname > POINTER$rec THEN
          PASSED$rec = '1'B ;
      END ;
      recindx = 1 ;
```

Case 3: Input, Nonsequential (ISAM), Keyed record.

Verify that the declaration

```
      DCL FOUND$rec BIT(1) ;
```

has been issued. Then issue the code:

```
      FOUND$rec = '1'B ;
      ON KEY (file_suff) FOUND$rec = '0'B ;
      READ FILE(file_suff) INTO(recbuf)
               KEY(POINTER$rec) ;
      recindx = 1 ;
```

Case 4: Output, Sequential Record.

Issue the following code:

```
      recindx = 1 ;
```

Call PACK procedure to pack its fields into the record buffer. Then issue the code:

```
      WRITE FILE(file_suff) FROM(recbuf) ;
```

Case 5: Output, Nonsequential, Keyed and an Update Record (both NEW and OLD specified)

Issue the following code:

```
        recindx = 1 ;
    Call PACK procedure to pack its fields into the record buffer.  Then
    issue the code:
        REWRITE FILE(file_suff) FROM(recbuf)
            KEY(POINTER$rec) ;
```

Case 6: Output, Nonsequential and Keyed Record.

```
    Issue the following code:
        recindx = 1 ;
    Call PACK procedure to pack its fields into the record buffer.  Then
    issue the code:
        WRITE FILE(file_suff) FROM(recbuf)
            KEY(POINTER$rec) ;
```

7.6 PACKING AND UNPACKING

After a record is read we unpack its fields from the record buffer and place them in the respective declared structures. Similarly before a record is written we pack its fields into the record buffer. The data movement is performed by individual transfers of fields. The transfer statements may be interleaved with other statements which control the iteration over respective fields' dimensions. The transfer instructions for unpacking are generated elsewhere, in conjunction with the schedule elements associated with the input field nodes. The code for packing an output record is generated in GENIOCD and inserted right before the record buffer is to be written out.

7.6.1 PACK - PACKING THE OUTPUT FIELDS

The procedure PACK is called by GENIOCD in the case of an output record. It accepts a node number (NODE$) as input. It checks the type of the node NODE$. If the node is a field, it calls DO_FLD to generate the code for packing. Otherwise, it considers in turn each descendant of the node NODE$. For each descendant D it calls PACK1(D) recursively.

PACK1: This procedure generates code for packing a node which may or may not repeat.

1. If the node is a repeating group or a field we get the termination criterion of the repetition.
   1.1 Open a loop: Call procedure GENDO to generate the DO-statement for opening the loop.
   1.2 Call the subprocedures PACK to issue code for packing a single element of the node.
   1.3 Call procedure GENEND to generate the code for terminating the loop.
2. If the node is not repeating then:
   Call procedure PACK to generate the code for packing all the constituent members of this node.

DO_FLD: This procedure is responsible for producing code to pack a field F into record buffer. It uses the procedure FIELDPK to generate the following code.

```
        SUBSTR(recbuf,recindx,lenstring) = F ;
        recindx = recindx+lenstring ;
```

FIELDPK is described further below.

7.6.2 GENITEM - UNPACKING THE INPUT FIELDS

This procedure is called to generate code for unpacking information from an input buffer to an input field. GEN_NODE calls GENITEM upon scanning a schedule element of an input field. GENITEM accepts as input the node number in the schedule element. The READ statement for reading the record to a buffer is generated by GENIOCD when the record node is scanned. GENITEM first finds for a record R the names of the input buffer RS and the packing counter RINDX. Next, GENITEM calls an auxiliary procedure FIELDPK, which generates the code for unpacking.

The GENITEM procedure is as follows:

1. Determine the name of the record containing the current field. Let it be rec. Then we construct a buffer name: rec_S and a buffer index name rec_INDX. Let the field's name be in the variable "field".
2. If the corresponding field in the next record is referenced, then call FIELDPK to unpack the field from the file buffer.
3. Call FIELDPK to generate the code for unpacking the field from the record buffer.

7.6.3 FIELDPK - PACKING AND UNPACKING FIELDS

The procedure FIELDPK produces the code for both the packing and unpacking operation. Input parameters are the field name, buffer name, record index name, and a code (CASE) to indicate whether the field has a NEXT prefix.

1. If the length type of the field is fixed, i.e. specified in the data description statements, we compute its length directly. If the field's type is 'C', 'N', or 'P', denoting respectively character, numeric or picture, we take the declared length. Otherwise we will compute the length of the field in bytes from its declared length and type. The string representing the length is stored in "lenstring".
2. If the length of the field was declared by specifying lower and upper bounds we check that there exists a control variable of the form LEN.field for this field. If none exists we issue the error message:
   ```
   FIELDPK: NO LENGTH SPECIFICATION FOR THE FIELD-field.
   ```
3. If a LEN.field control variable is found we set:
   ```
   lenstring = LEN.field
   ```
   The byte-length of the field will be computed during run time.
4. If the field is an input field we generate the instruction:
   ```
   UNSPEC(field) = SUBSTR(rec_S,rec_INDX,lenstring);
   ```
   If the same field in the next record is referred in the specification, we will unpack the file buffer to get the

corresponding field in the next record. For output field we generate:

```
        SUBSTR(rec_S,rec_INDX,lenstring) = UNSPEC(field) ;
```

Here "field" is the name properly subscripted and "lenstring" is the length specification. If the field is of type 'C', the UNSPEC qualifications will be omitted.

5. If the CASE code indicates that the field name does not have prefix NEXT then we generate the following code to update the buffer index:

```
        rec_INDX = rec_INDX+lenstring ;
```

There is no need to update recINDX if the unpacking is for a NEXT prefixed field.

7.7 GENERATING THE PROGRAM ERROR FILE

If a program error condition is induced during the execution of the generated program, then an input record, read during the iteration execution when the program error was induced is written to an error file, ERRORF. The required code for writing the bad input record to the error file is generated by the routines CODEGEN and GENIOCD. For example, the following PL/I code is included in PL1ON file:

```
        ON ERROR BEGIN ;
          WRITE FILE(ERRORF) FROM($ERROR_BUF) ;
          GO TO $ERR_LAB ;
        END ;
```

After the GENIOCD generate the code to read a record from an input file it also generates a statement to copy the input record into $ERROR_BUF.

7.8 GPL1DCL - GENERATING PL/I DECLARATION

This procedure generates the declarations for the data nodes declared by the user and those added by the system. As noted previously, some declarations are also generated by other procedures during the code generation.

The main part of GPL1DCL is as follows:

1. For each file F in the specification (available from the list FILIST) call

```
        DECLARE_STRUCTURE(F)
```

   to declare F and all its descendants.
2. For each node N in the specification which is an interim variable or a control variable, call

```
        DECLARE_STRUCTURE(N)
```

3. For each subscript which has been used, issue the declaration:

```
        DCL subname FIXED BIN ;
```

7.8.1 DECLARE_STRUCTURE - DECLARING A STRUCTURE

This procedure is called by GPL1DCL. The input is a file node number. It declares the entire file structure. It issues the declarative: DECLARE, and then proceed to call DCL_STR(N,1,0).

7.8.1.1 DCL_STR(N, LEVEL, SUX)

This recursive procedure produces a declaring-clause for each node N in the structure. 'LEVEL' is the current level in the structure. SUX is a termination criterion stating whether there is a next node on the same level (younger brother) or a descendant.

1. Some Preliminary transformations are made on the declared node names.
   1.1 File names of the form NEW.F and OLD.F are modified to NEW_F and OLD_F respectively.
   1.2 The group names, record names, or field names are reduced to their stem (removing prefixes).
2. For control variables the resulting declaration is:

   For SIZE, and LEN names:

   name FIXED BIN,

   while for all other names:

   name BIT(1).
3. The declaration includes in general the following items:

   LEVEL - The component level.

   Name - The declared name.

   Repetition - The number of physical storage elements.

   Type - The data type.

   The data type is determined as follows:

For character fields	- CHAR(len) [VARYING]
For numeric fields	- PIC '99....9'
For picture fields	- PIC 'picture'
For fixed binary	- BIN FIXED(len, scale)
For fixed decimal	- DEC FIXED(len,scale)
For binary floating	- BIN FLOAT(len)
For decimal floating	- DEC FLOAT(len)

   In the above 'len' is the specified or default length for the field. The VARYING option is taken if the length is specified (for strings) by a minimal length and a maximal length.

   Repetition is defined in STOTYP of the node subscripts of the fields. If an array dimension is virtual we omit the repetition indicator. If an array dimension is a window of width k+1, the repetition is set to k+1. Otherwise, the array dimension must be a physical dimension. The node subscript list of the field node is scanned, and the repetition indicators for array dimensions are concatenated and put into a variable REP. If R is not an empty string, we will append the string '(REP)' after the declared field name.
4. For each of the descendants of the node M, call DCL_STR(M,LEVEL+1,termination) recursively.

7.9 CGSUM - CODE GENERATION CONCLUSION

CGSUM has the task of concluding the code generation phase. First, the different files with the generated PL/I program (PL1DCL, PL1ON, PL1EX) are merged into one PL/I file (PL1PROG) which can be subsequently compiled. Secondly, a Code Generation Summary Report is written which lists the PL/I program. While the PL/I listing would not be of much use to the average MODEL user, it is of interest to the more sophisticated user and can serve the system programmer for insight or debugging of the MODEL system.

BIBLIOGRAPHY

ADAA 79
Preliminary ADA Reference Manual.
SIGPLAN Notices 14(6): Part A, June, 1979.

ADAB 79
Rationale for the Design of the ADA Programming Language.
SIGPLAN Notices 14(6): Part B, June, 1979.

AHJO 76
Aho, A.V., and Johnson, S.C.
Optimal Code Generation for Expression Trees.
JACM 23(3):488-501, July, 1976.

AHJO 77
Aho, A.V., and Johnson, S.C.
Code Generation for Expressions with Common Subexpressions.
JACM 24(1):146-160, January, 1977.

AHU 74
Aho, A.V., Hopcroft, J. E., and Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Reading, Mass., Addison-Wesley, 1974.

AHUL 78
Aho, A.V., and Ullman, J.D.
Principles of Compiler Design.
Addison-Wesley, 1978.

ASWA 77
Ashcroft, E.A., and Wadge, W.W.
Lucid, A Non-procedural Language with Iteration.
CACM 20(7):519-526, July, 1977.

BAFI 79
Bauer, J., and Finger, A.
Test Plan Generation Using Formal Grammars.
Fourth International Conference on Software Engineering, pp. 425-432. September, 1979.

BAKA 76
Barstow, D.R., and Kant, E.
Observations on the Interaction Between Coding and Efficiency Knowledge in the PSI Program Synthesis System.
Second International Conference on Software Engineering, pp. 19-31. October, 1976.

BOKP 76
Bobillier, P. A., Kahan, B. C., and Probst, A. R.
Simulation with GPSS and GPSS V.
Prentice-Hall, 1976.

BRSE 76
Bruno, J., and Sethi, R.
Code Generation for a One-Register Machine.
JACM 23(3):502-510, July, 1976.

BUDA 77
Burstall, R. M., and Darlington, J.
A Transformation System for Developing Recursive Programs.
JACM 24(1):44-67, January, 1977.

CHTH 79
Cheatham, T. E., Townley, J. A., and Holloway, G. H.
A System for Program Refinement.
Fourth International Conference on Software Engineering, pp. 53-62. September, 1979.

CHHT 81
Cheatham, T. E., Holloway, G. H., and Townley, J. A.
Program Refinement by Transformation.
Fifth International Conference on Software Engineering, pp. 430-437. March, 1981.

DAMN 70
Dahl, O. J., Myhrhaug, B., and Nygaard, K.
The SIMULA 67 Common Base Language.
Publication S-22, Norwegian Computing Center, Oslo, 1970.

DAVI 79
Davis, A., et al.
RLP: An Automated Tool for the Automatic Processing of Requirements.
Proceedings of COMPSAC 1979, pp. 289-299.

DAVI 80
Davis, A.
Automating the Requirements Phase: Benefits to Later Phases of the Software Life-Cycle.
Proceedings of COMPSAC 1980, pp. 42-48.?

FELD 72
Feldman, J.A.
Automatic Programming.
Technical Report STAN-CS-72-255, Stanford University, February, 1972.

GEMS 77
Geschke, C.M., Morris, J.H., and Satterthwaite, E.H.
Early Experience with MESA.
CACM 20(8):540-553, August, 1977.

GOKH 81
Gokhale, M.
Data Flow Analysis of Non-procedural Specifications.

Proposal for PhD research. University of
   Pennsylvania, July, 1981.

GRAH 80
   Graham, S. L.
   Table-Driven Code Generation.
   IEEE Computer, 13(8):25-34, August, 1980.

GREB 81
   Greenburg, R.
   Simultaneous Equations in the MODEL system with an
      Application to Econometric Modelling
   Master Thesis, University of Pennsylvania, October, 1981.

GREE 69
   Green, C.
   Application of Theorem Proving to Problem Solving.
   First Joint Conference on Artificial Intelligence,
      pp. 219-239. 1969.

GREE 77
   Green, C.
   The Design of the PSI Program Synthesis System.
   Second International Conference on Software
      Engineering, pp. 4-18. IEEE Press, October, 1976.

HHKW 77
   Hammer, M., Howe, W.G., Kruskal, V.J.,
      and Wladawsky, I.
   A Very High Level Programming Language for Data
      Processing Applications.
   CACM 20(11):832-840, November, 1977.

IVER 62
   Iverson, K.E.
   A Programming Language.
   Wiley, New York, 1962.

KESC 75
   Kennedy, K., and Schwartz, J.
   An Introduction to the Set Theoretical Language
      SETL.
   Comp. and Maths with Appls, Vol. 1, pp. 97-119.
      Pergamon Press, 1975. Great Britain.

KKPL 81
   Kuck, D. J., Kuhn, R. H., Padua, D. A.,
      and Wolfe, M.
   Dependence Graphs and Compiler Optimizations
   In Proceedings of the ACM National Conference,
      Pages 207-218. ACM, 1981.

LCHN 80
   Leverett, B. W., Cattell, R. G., Hobbs, S. O.,
      Newcomer, J. M., Reiner, A. H., Schatz, B. R.,

and Wulf, W. A.
An Overview of the Production-Quality-Compiler-Compiler Project.
IEEE Computer, 13(8):38-49, August, 1980.

LEWA 74
Lee, R.C.T., and Waldinger, R.J.
An Improved Program Synthesis Algorithm and its Correctness.
CACM 17(4):211-217, April, 1974.

LSAS 77
Liskov, B., Snyder, A., Atkinson, R., and Shaffert, C.
Abstraction Mechanisms in CLU.
CACM 20(8):564-574, August, 1977.

LYNC 69
Lynch, H.
ADS: A Technique in System Documentation.
Database, 1, 1 (Spring 1969), pp. 6-18.

MANN 71
Manna, Z.
Toward Automatic Program Synthesis.
CACM 14(3): 151-165, March, 1971.

MCCU 77
McCune, B.P.
The PSI Program Model Builder: Synthesis of Very High-Level Programs.
SIGPLAN Notices 12(8):130-138, August, 1977.

NUKO 76
Nunamaker, J.F., and Konsynski, B.R.
Computer-Aided Analysis and Design of Information System.
CACM 19(12):674-687, December, 1976.

NUNA 71
Nunamaker, J.F.
A Methodology for the Design and Optimization of Information Processing Systems.
SJCC, pp. 283-294, AFIPS Press, 1971.

PARN 72
Parnas, D.L.
On the Criteria to be Used in Decomposing Systems into MOdules.
CACM 15(12):1053-1058, December, 1972.

PHIL 77
Phillips, J.V.
Program Inference from Traces Using Multiple Knowledge Sources.

Fifth Joint Conference on Artificial Intelligence,
pp. 812. August, 1977.

PNPR 80
Pnueli, A, and Prywes, N.
Operations on Arrays and Data Structures.
Technical Report, MCS-79-0298, University of
Pennsylvania, 1980.

PNPR 81
Pnueli, A, and Prywes, N.
Distributed Processing in the MODEL System with an
Application to Econometric Modelling.
Technical Report, MCS-79-0298, University of
Pennsylvania, 1981.

RAMI 73
Ramirez, J.A.
Automatic Generation of Data Conversion Programs
Using A Data Definition Language.
PhD thesis, University of Pennsylvania, 1973.

RIN 76
Rin, N.A.
Automatic Generation of Data Processing Programs
from a Non-procedural Language.
PhD thesis, University of Pennsylvania, 1976.

SANG 80
Sangal, R.
Modularity in Non-procedural Languages Through
Abstract Data Types.
PhD thesis, University of Pennsylvania, July, 1980.

SCAN 72
Schneck, P. B., and Angel, E.
FORTRAN to FORTRAN Optimizing Compiler.
The Computer Journal (British Computer Society)
16(4):322-329, 1972.

SCH 75
Schwartz, J. T.
Optimization of Very High Level Languages-I.
Journal of Computer Languages, Vol. 1, pp. 161-194.
Pergamon Press, 1975. Northern Ireland.

SCHA 73
Schaefer, M.
A Mathematical Theory of Global Program
Optimization.
Prentice-Hall, New Jersey, 1973.

SCHW 75
Schwartz, J. T.
Optimization of Very High Level Languages-II.

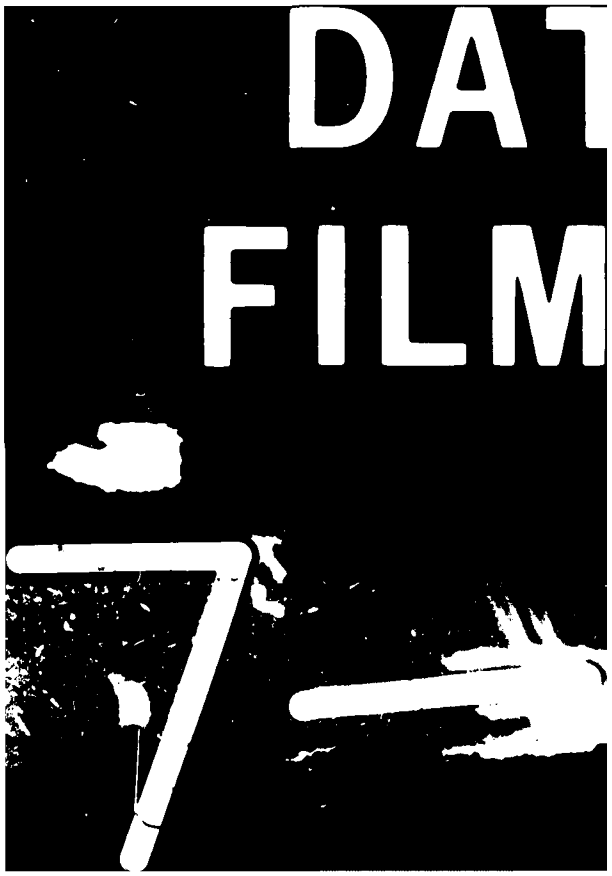
Journal of Computer Languages, Vol. 1, pp. 197-218. Pergamon Press, 1975. Northern Ireland.

SHAS 78
Shastry, S.
Verification and Correction of Non-procedural Specification in Automatic Generation of Programs.
PhD thesis, University of Pennsylvania, 1978.

SZHM 77
Szolovits, P., Hawkinson, L., and Martin, W.
An Overview of OWL, A Language for Knowledge Representation.
Technical Report MIT-LCS-TM-86, MIT, June, 1977.

WALE 69
Waldinger, R.J., and Lee, R.C.T.
PROW: A Step Towards Automatic Program Writing.
First Joint Conference on Artificial Intelligence, pp. 241-252. 1969.

WEGN 79
Wegner, P.
Research Directions in Software Technology.
MIT Press, 1979.


DAT
FILM
7